CASE STUDIES ON OPTIMIZING ALGORITHMS FOR GPU ARCHITECTURES

Shawn Daniel Brown

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the College of Arts & Sciences.

Chapel Hill
2015

Approved By:

Anselmo Lastra

Dinesh Manocha

Lars Nyland

Jan Prins

Jack Snoeyink

ABSTRACT

SHAWN DANIEL BROWN: Case Studies on Optimizing Algorithms for GPU Architectures
(Under the direction of Jack Snoeyink)

Modern GPUs are complex, massively multi-threaded, and high-performance. Programmers naturally gravitate towards taking advantage of this high performance for achieving faster results. However, in order to do so successfully, programmers must first understand and then master a new set of skills – writing parallel code, using different types of parallelism, adapting to GPU architectural features, and understanding issues that limit performance.

To help GPU programmers become productive more quickly, this dissertation introduces three *data access skeletons* (DASks) – Block, Column, and Row -- and two *block access skeletons* (BASks) – Block-By-Block and Warp-by-Warp. Each “skeleton” provides a high-performance implementation framework that partitions data arrays into data blocks and then iterates over those blocks. Programmers must still write “body” methods on individual data blocks to solve their specific problem. These skeletons provide efficient machine dependent data access patterns for use on GPUs. DASks group *n* data elements into *m* fixed size data blocks. These *m* data block are then partitioned across *p* thread blocks using a 1D or 2D layout pattern. Generic programming techniques are applied to the fixed size data blocks to enable performance experiments for different types of parallelism – instruction-level parallelism (ILP), data-level parallelism (DLP), and thread-level parallelism (TLP).

These different DASks and BASks are introduced using a simple memory I/O (Copy) case study. Three additional case studies – Reduce/Scan, Histogram, and Radix Sort -- demonstrate DASks and BASks in action on parallel primitives and also provide more valuable performance lessons.

To Helene, my love, my rock, my soul mate your love has helped support and guide me through these many tough years working on my PhD. You showed great wisdom, love, and courage during some turbulent times within our family.

Proverbs 31:10-31

"Who can find a virtuous woman? For her price is far above rubies. … She opens her mouth with wisdom; and in her tongue is the law of kindness. … Her children arise up, and call her blessed; her husband also, and he praises her."

ACKNOWLEDGMENTS

A PhD is a long hard struggle over many years. It takes a lot of support from others to make it through the many false starts, failures, and setbacks required to complete such a huge undertaking. I am very grateful to my mentor, Jack Snoeyink, for his patient guidance, amazing insights, and unwavering support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ALU	**A**rithmetic **L**ogic **U**nit
All-kNN	All ***k*** **N**earest **N**eighbor (search)
All-NN	**A**ll **N**earest **N**eighbor (Search)
AMP	**A**ccelerated **M**assive **P**arallelism (Microsoft)
ANN	**A**pproximate **N**earest **N**eighbor (Search)
API	**A**pplication **P**rogram **I**nterface
b, B	Bit or boolean {0|1} or {*true*|*false*}
BASk	**B**lock **A**ccess **Sk**eleton
BFS	**B**reath **F**irst **S**earch
bid	**b**lock **id** (uniquely identifies thread block within grid)
BLP	**B**it-**L**evel **P**arallelism
BPS	**B**its **P**er **S**econd
BSP	**B**inary **S**pace **P**artition (tree)
BYTE	Derived from ‘bit’ and ‘bite’ (8-bits of data)
CC	**C**lock **C**ycle or **C**ompute **C**apability
CPI	(average) **C**ycles **P**er **I**nstruction (for entire algorithm)
CPU	**C**entral **P**rocessing **U**nit
CTA	**C**ooperative **T**hread **A**rray (Grid of Thread Blocks)
CUDA	**C**ompute **U**nified **D**evice **A**rchitecture (NVIDIA)
DASk	**D**ata **A**ccess **Sk**eleton
DBS	**D**ata **B**lock **S**ize
DDR	**D**ouble **D**ata **R**ate memory type (as in DDR3-RAM)
DFS	**D**epth **F**irst **S**earch
DLP	**D**ata-**L**evel **P**arallelism
DMA	**D**irect **M**emory **A**ccess

DWORD	**D**ouble **Word** (32-bits of data, 4 bytes)
EX	**Ex**ecute cycle (classic MIPS pipeline)
FMA	**F**used **M**ultiply **A**dd instruction (in ISA)
FPU	**F**loating **P**oint **U**nit
G	**G**iga (one billion)
GB	**G**iga-**B**yte (one billion bytes)
GM	**G**lobal **M**emory
GPU	**G**raphics **P**rocessing **U**nit
HPC	**H**igh **P**erformance **C**omputing
ID	**I**nstruction **D**ecode cycle (classic MIPS pipeline)
IDE	**I**nteractive **D**evelopment **E**nvironment
IF	**I**nstruction **F**etch cycle (classic MIPS pipeline)
II	(total) **I**nstructions **I**ssued (for entire algorithm)
ILP	**I**nstruction-**L**evel **P**arallelism
IPC	(average) **I**nstructions (retired) **P**er **C**ycle
ISA	**I**nstruction **S**et **A**rchitecture
K	**K**ilo (one thousand)
KB	**K**ilo-**B**yte (one thousand bytes)
*k*d	*k*-**D**imensional
*k*d-tree	A generalized binary tree used for spatial searching
*k*NN	***k*** **N**earest **N**eighbor (search)
KSHS	**K**ogge-**S**tone-**H**illis-**S**teele prefix-sum (scan)
LSD	**L**east **S**ignificant **D**igit
M	**M**ega (one million)
MB	**M**ega-**B**yte (one million bytes)
MEM	**Mem**ory access cycle (classic MIPS pipeline)

MISD	**M**ultiple-**I**nstruction, **S**ingle **D**ata
MIMD	**M**ultiple-**I**nstruction, **M**ultiple **D**ata
MIPS	**M**icroprocessor (without) **I**nterlocked **P**ipeline **S**tages
MPI	**M**essage **P**assing **I**nterface
MPMD	**M**ultiple-**P**rogram, **M**ultiple-**D**ata
MSD	**M**ost **S**ignificant **D**igit
NN	**N**earest **N**eighbor (Search)
NOP	**N**o **Op**eration
octree	A 3D hierarchical spatial searching data structure
OpenCL	**Open C**omputing **L**anguage (Khronos Group)
PTX	**P**arallel **T**hread E**x**ecution assembly code (NVIDIA)
quadtree	A 2D hierarchical spatial searching data structure
QNN	**Q**uery **N**earest **N**eighbor (search)
QWORD	**Q**uad **Word** (64-bits of data, 8 bytes)
RB	**R**each **B**ack (convert inclusive to exclusive scan)
RAM	**R**andom-**A**ccess **M**emory
RAW	**R**ead **A**fter **W**rite (data hazard)
RISC	**R**educed **I**nstruction **S**et **C**omputer
RNN	**R**ange (**N**earest **N**eighbor) Query Search
RAM	**R**andom **A**ccess **M**emory
SM	**S**hared **M**emory
Sh. Mem.	**Sh**ared **Mem**ory
SIMD	**S**ingle-**I**nstruction, **M**ultiple-**D**ata
SIMT	**S**ingle-**I**nstruction, **M**ulti-**T**hreaded
SISD	**S**ingle-**I**nstruction, **S**ingle-**D**ata
SM	**S**treaming **M**ulti-core (Multi-processor)

SMX	**S**treaming **M**ulti-core E**x**tended (Multi-processor)
SMP	**S**imultaneous **M**ulti-**P**rocessor (machine)
SMT	**S**imultaneous **M**ulti-**T**hreading
SoC	**S**ystem **o**n a **C**hip (Modern CPUs)
SP	**S**calar **P**rocessor (within SM)
SPMD	**S**ingle-**P**rogram, **M**ultiple-**D**ata
SPMT	**S**ingle-**P**rogram, **M**ulti-**T**hreaded
T	**T**era (one trillion)
TB	**T**era-**B**yte (one trillion bytes)
TBS	**T**hread **B**lock **S**ize
TC	**T**otal **C**ycles (for entire algorithm)
tid	**t**hread **id** (uniquely identifies thread within block)
TLP	**T**hread-**L**evel **P**arallelism
TRISH	**T**hreaded **R**egister **I**nterleaved **S**trided **H**istogram (method)
VP	**V**ector-level **P**arallelism (Vectorization, or **V**ector **P**rocessing)
WAR	**W**rite **A**fter **R**ead (data hazard)
WAW	**W**rite **A**fter **W**rite (data hazard)
WB	**W**rite-**B**ack cycle (classic MIPS pipeline)
WORD	**Word** (16-bits of data, 2 bytes)

LIST OF SYMBOLS

$\odot$	Unary Transform operator, $b = \odot\, a = \odot\,(a)$
$\oplus$	Binary Transform operator, $c = a \oplus b = \oplus(a, b)$
A_n	Array of n data elements
C	Number of columns, if the number of rows (R) is fixed it can be computed as $C = \lceil m/R \rceil$
c_i	The ith bin counter, taken from $[0,d)$ bins in a count histogram (Radix Sort).
CTA layout	Sizes and IDs of CTA grid and thread blocks respectively. Even though the size parameters can be taken directly from CTA runtime parameters they are usually indirectly set as compile-time constants for improved performance and to support generic programming. The position IDs are always runtime parameters.
gw	*Grid Width*, AKA `gridDim.x`
gh	*Grid Height*, AKA `gridDim.y`
gl	*Grid Length*, AKA `gridDim.z` (currently unused)
bw	*Block Width*, AKA `blockDim.x`
bh	*Block Height*, AKA `blockDim.y`
bl	*Block Length*, AKA `blockDim.z`
bx	Position within [0, *gw*), AKA `blockIdx.x`
by	Position within [0, *gh*), AKA `blockIdx.y`
bz	Position within [0, *gl*), AKA `blockIdx.z`
tx	Position within [0, *bw*), AKA `threadIdx.x`
ty	Position within [0, *bh*), AKA `threadIdx.y`
tz	Position within [0, *bl*), AKA `threadIdx.z`
CTA params	Derived parameters from the CTA layout parameters, these are usually computed as runtime parameters.
b	Block Size, assumes block is 1D and small ($b \leq 1024$),
	Typically taken directly from `blockDim.x`
g	Grid Size, assumes grid is 1D and small ($g \leq 1000$),

	Taken directly from either `gridDim.x` or `*.y`
bid	Unique 1D *block ID* (thread block within a grid), derived (mapped) from ‹*gw*,*gh*,1› and ‹*bx*,*by*,*bz*›
tid	Unique 1D *thread ID* (thread within a thread block), derived (mapped) from ‹*bw*,*bh*,*bl*› and ‹*tx*,*ty*,*tz*›
warpCol	Relative thread position within current thread warp in range [0, *WarpSize*) = [0, 31). Computed as *warpCol* = *tid* % *WarpSize* = *tid* % 32 or alternately as *warpCol* = *tid* & (*WarpSize*-1) = *tid* & 31.
warpRow	Relative warp position within current thread block in range [0, *nWarps*). Computed as *warpRow* = *tid* / *WarpSize* = *tid*/32 or alternately as *warpRow* = *tid* >> $\log_2$(*WarpSize*) = *tid*>>5.
d	Number of dimensions for d-dimensional points
d	Digit, represents a small fixed-size numeric range [0, *d*). A number is represented by a string of digits (Radix Sort).
d	Number of bin counters (Count Histogram, Radix Sort)
$D(n), D$	Depth, Steps, Number of parallel stages for n data elements
DBS	Data Block Size, AKA Data (Work) per block, computed as *DBS* = *nWork***nWarps***WarpSize* (= *nWork***TBS*).
dist(*p*,*q*)	A distance metric between two points, Euclidean distance as *dist*(*p*,*q*)=$\sqrt{(q_1-p_1)^2+(q_2-p_2)^2+\cdots+(q_d-p_d)^2}$
dv	Digit value, specific number taken from range [0, *d*). A specific number is represented by a string of 1 or more specific digit values from most to least significant.
DWord	32-bit data-element (4 bytes, 2 words)
f	Fanout in a circuit (hardware adders)
G	Cycles to transfer data between global memory and registers (400-800 cycles on Fermi, 200-400 on Kepler)
h_i	The *i*th bin count (matching the sub-range r_i)

$\mathbb{I}$	Identity element, i.e., $\mathbb{I}$ is a data element under some binary operator $\oplus$ such that $a = a\oplus\mathbb{I} = \mathbb{I}\oplus a$ for all $a \in \mathbb{U}$
IS	The linear *ideal speedup* of a parallel computation, computed as *IS* = *W(n)*/*p*.
k	Pipelined instruction-length (18-22 cycles on Fermi, 9-11 cycles on Kepler)
k	Number of nearest neighbors to find (kNN search)
k	Number of work items per-thread
k	Number of thread collisions in a *k*-way bank conflict.
k	Number of binning passes (Radix Sort)
k	The maximum number of digits in a key (Radix Sort), computed as $k = \lceil\log_d m\rceil$
l	Logic levels in a circuit (hardware adders)
n	Number of data elements in a run, warp, or array
n	Number of points (objects) in a search set (S)
nWarps	C++ parallelism parameter that tracks the number of thread warps per thread block, typically set in the range [1-8] but can go as large as 32 warps on current GPU architectures.
nWork	C++ parallelism parameter that tracks the number of work-items (data elements) assigned to each thread, typically set in the range [1-8] but can go larger as needed.
m	Number of bins (frequency counts in histogram)
m	Number of data blocks, $m = \left\lceil\frac{n}{DBS}\right\rceil$
m	Number of points (objects) in a query set (Q).
m	Maximum number value taken from a large fixed-size numeric range [0, *m*).
$O(n)$	Big "O" Notation, Asymptotic complexity of an algorithm
$\mathbb{P}$	Parallelism (work over depth), $\mathbb{P} = \frac{\boldsymbol{W(n)}}{\boldsymbol{D(n)}}$
p	Number of parallel processors (multi-cores)
p	Number of parallel threads
point	A *d*-dimensional point (optionally representing an object) as

	$\langle x, y, \ldots \rangle$
Q	*Query set*, set of points (objects) to find closest neighbors to
QR	*Query region*, a set of regions QR_i (typically d-dimensional hyperboxes or hyperspheres) to find points (objects) that are contained in or covered by the query regions.
R	Range $R = [min, max]$ or $R = [min, max)$
R	Number of rows, if the number of columns (C) is fixed it can be computed as $R = \lceil m/C \rceil$
r_i	The ith sub-range, $r_i = [a_i, a_{i+1})$ from a larger range R
r_i	The ith row from p rows in a partitioned data set
r_i	The ith run from p runs in a partitioned data block
r_i	The ith digit run, where the chosen digit from each key in the run matches the digit value i in the range $[0, d)$ (Radix Sort)
rl	Run length, typically in range [1-8].
S	Cycles to transfer data between shared memory and registers (40-80 on Fermi, 20-40 on Kepler)
S	*Search set*, set of points (objects) to be searched
$S(n), S$	Steps, Number of parallel stages for n data elements
	Equivalent to concept of $Depth = D(n) = D$
s_i	The ith run start, taken from $[0,d)$ bins in a start histogram (Radix Sort).
Speedup	Speedup, computed as $S = \frac{Time_{old}}{Time_{new}}$
…Serial	Serial Speedup, computed as $SS = \frac{Time_{baseline}}{Time_{improved}}$
Parallel	Parallel Speedup, computed as $SS = \frac{Time_{serial}}{Time_{parallel}}$
$SR\langle n \rangle$	A *Serial Reduce* on a run of length n
$SS\langle n \rangle$	A *Serial Scan* on a run of length n
$SU\langle n \rangle$	A *Serial Update* on a run of length n (add prefix to run)
t	Number of threads
t	Number of search results (NN searches)
T	Wire tracks in a circuit (hardware adders)

T_1	Parallel running time on one processor for a parallel algorithm, equivalent to the concept of $Work = W(n) = W$
TBS	Thread Block Size, AKA threads per block, computed as *TBS=nWarps*WarpSize*
TC	Total Cycles, computed as $TC = II/IPC$.
T_{CPU}	Serial running time on one processor for serial algorithm
T_p	Parallel running time for a fixed number of p processors bounded by **Brent's Theorem:** $max\left(\frac{W}{p}, D\right) \leq T_p \leq \frac{W}{p} + D$
T_∞	Parallel running time for an infinite number of processors, equivalent to the concept of $Depth = D(n) = D$
WarpSize	C++ parallelism parameter, used to track threads per thread warp. This value is currently fixed at 32 on current GPU architectures, but could change in future architectures.
$W(n), W$	Work, Total parallel work across p processors for n elements
	Limit of running time for 1 processor
WR‹n›	A *Warp Reduce* on a data run of length *n* = [2, 4, …, 32].
WR‹n›	A *Warp Scan* on a data run of length *n* = [2, 4, …, 32]

1.0 Introduction

Someone who wants to adapt a sequential program to a graphics processing unit (GPU) for better performance must learn to deal with a number of challenging problems quickly and without much training. In order to ease this process and help GPU programmers become productive more quickly, I have developed several skeletons that can be modified to fit their environment without having to come up with new code on their own. This thesis explains what these skeletons are and how they can be modified to solve real-life challenges. However, before I can explain how they help address those challenges, I must explain what those challenges are.

The next section attempts to explain in a nutshell why GPU programming is hard. New terminology employed in this section will be explained later on in context as this thesis unfolds.

1.1 The GPU Performance Challenge:

The Challenge: GPU programmers are faced with several, complex challenges that directly affect performance. They must first understand and select serial algorithms that they can depend upon to solve their specific problems. They then must convert each single-threaded serial algorithm into an equivalent massively multi-threaded parallel algorithm[1] that is both correct and robust. They must carefully implement their multi-threaded solutions to prevent resource contention between threads that prevents problems such as race conditions, dead-lock, live-lock, starvation, etc. If the programmer is not careful, the overhead required to prevent resource contention can overwhelm the amount of useful work, bottlenecking performance.

[1] Parallel algorithms are discussed in the following books and papers (Atallah et al, 2010; Blelloch and Maggs, 1996; Dongarra et al, 2003; Hillis and Steele, 1986; Hwu, editor, 2011 and 2012; Miller and Boxer, 2013; Rauber and Rünger, 2010)

In order to achieve high performance, GPU programmers must understand GPU architecture[2]: how the 2-level compute system operates, how batches of threads are mapped onto batches of simple cores, how the memory hierarchy consists of many memory types and behaviors, and so forth. They also need to understand the resource issues imposed by the GPU architecture and the trade-offs needed in order to avoid bottlenecks. Since the hardware supports pipelining, they must understand *Instruction-level parallelism* (ILP) and how to rewrite code to unlock ILP performance. Since the hardware supports *single-instruction-multiple-data* (SIMD)[3], they must understand *Data-level parallelism* (DLP), data partitioning, coalescence, and load-balancing. Since the hardware supports multi-threading, they must understand *Thread-level parallelism* (TLP), latency, warps in flight, occupancy, and related resource limitations. They must worry about how to transfer data efficiently between the *central processing unit* (CPU) and GPU. Given a large parameter space of apparently equal and valid choices, they must explore these many choices to help select the best parameters for optimal balanced performance on a particular GPU device. Most of all, GPU programmers must be creative and willing to re-design and re-implement their solutions in order to achieve their desired performance goals.

Rising to the Challenge: Even though achieving high performance GPU algorithms for non-trivial algorithms is hard, solving complex scientific problems on massive data sets is worth the extra effort. The payoff is seeing solutions that used to take hours or days of computing time now finish in seconds or minutes.

Solid performance on the GPU is achieved by

1) Picking good algorithms
2) Using parallel programming concepts
3) Adapting to the GPU architecture
4) Eliminating performance issues

[2] GPU architecture is discussed in the following books and papers (Buck et al, 2004; Garland and Kirk, 2010; Göddeke et al, 2011; Hennessey and Patterson, 2012; Hwu, editor; 2011 & 2012; Nickoos and Dally, 2010; NVIDIA 2010 Fermi; NVIDIA, 2012 Kepler; Owens et al, 2007; Patterson, 2009).
[3] As introduced in Flynn's Taxonomy (Flynn, 1972).

Finally, the programmer must then iterate on the previous four concepts until satisfied with performance as measured by some metric against baseline performance as shown in the following five case studies: *Memory I/O*; *Scan/Reduce*; *kd-tree*; *Histogram*; and *Radix Sort*.

1.1.1 Good algorithms

The entire point of computer science has been to solve big real-world problems by developing great data structures and solid algorithms Experienced programmers are well of broad paradigms[4] like *Divide and Conquer*, *Recursion*, *Greedy Algorithms*, *Dynamic Programming*, *Reductions*, *Randomized algorithms*, etc. The main point of these approaches is to minimize the real-world resource consumption (time, space, etc.) of algorithms using a concept called *Asymptotic Growth*, also known as “big-O” notation. “Big-O” notation allows programmers to broadly compare different solutions to the same problem. For example, an algorithm that takes linear time $O(n)$ is generally considered better than an algorithm that takes log-linear time $O(n \log n)$ or even quadratic time $O(n^2)$.

Experienced programmers have many data structures and algorithms in their tool-belt that allows them to creatively solve real-world problems with their many competing demands and constraints. Great programmers take their solutions a step further by making sure their code is efficient often achieving up to a ten-fold increase in performance over an algorithm as found in a class, book, or on the internet. There are many books, papers, lectures, and other resources that describe these efficient data structures or algorithms. Such a topic is outside the realm of this thesis. I will assume my readers have already found and picked good algorithms for their specific problem-space and are now struggling with getting their code working on a GPU. And that after they get their code working correctly, they will then want to write efficient code that achieves high performance.

[4] As described in various books on algorithms (Cormen et al, 2009; Edmonds, 2008; Miller and Boxer, 2013; Sedgewick, 1998; Skiena, 2008).

1.1.2 Key concepts for Parallel Computation in GPUs

GPU's achieve high performance by massive parallelism. Modern GPUs, like the GTX 580, 680 and Titan[5], contain hundreds, even thousands of processing cores. The GPU architecture supports multiple forms of parallelism[6]: instruction-level parallelism, thread-level parallelism, task-level parallelism, and data-level parallelism. ILP seeks to extract multiple independent instructions from sequential instruction streams, which can then be executed in parallel on multiple processing stages within each processing core. A single instruction stream is typically executed via a single thread on a multi-threaded CPU. TLP attempts to keep processors busy by rapidly switching between multiple instruction streams whenever the current thread stalls waiting on some other resource. TLP, also known as multi-threading, supports both task-level parallelism and data-level parallelism. For task-level parallelism, work is defined as an abstract unit of useful computation, functionality refers to useful functions, modules, sub-programs which can be grouped by common tasks such as graphics, audio, user-interface, etc.. *Task-Level Parallelism* divides work by functionality across multiple threads of execution with each subtask being mapped onto its own thread. Because of the heterogeneous nature of the subtasks, this form of parallelism works best on multi-core CPUs. For data-level parallelism (DLP), work is defined as by how many data elements (work items) are processed by each individual thread. DLP partitions a data array into smaller chunks with each thread being assigned its own individual chunk of data to process according to some data parallel function, kernel, or program. Each thread executes the same set of instructions but on different pieces of data. Because data-parallel code is largely homogenous across threads, DLP strongly favors GPUs with their massive number of simple cores arranged in SIMD layout. These different types of parallelism (ILP, TLP, and DLP) will be discussed in more detail in "Chapter 2 – Parallelism".

[5] As described by NVIDIA (NVIDIA, 2010 Fermi; NVIDIA, 2012 Kepler; NVIDIA, 2012, GTX 680)
[6] See the book Computer Architecture (Hennessey and Patterson, 2012) for more details on the various types of parallelism.

1.1.3 GPU Architecture

Adapting algorithms to GPU architectures is a challenge because of the many choices for thread organization, register assignment, memory layout, and synchronization. In this thesis I consider some of these choices using several case studies (*memcopy*, *scan/reduce*, *kd-tree*, *histogram*, and *RadixSort*) for processing massive amounts of data at throughputs rates that approach the hardware limits. In my experience, the paucity of choices for synchronization drives the initial design decisions. In addition, the memory layout determines what can be processed efficiently in parallel. Finally, there are many choices for processor organization that can be experimented with to approach peak performance. GPU architectures will be discussed in more detail in “Chapter 2 – Parallelism”.

1.1.4 GPU Performance Issues

GPU architectures focus on maximizing throughput across tens of thousands of threads, while CPU architectures concentrate on minimizing latency for a single task. Thus, there are many differences between these two architectures--including parallelism, multi-threading, and memory hierarchy. GPU programmers need to be aware of these differences to increase parallel throughput in their own GPU implementations. Although, some GPU hardware features can help performance, other GPU hardware features can hinder performance. Many of these performance issues related to GPU hardware will be discussed in more detail in “Chapter 3- Performance and Issues”.

1.2 Data Access Skeletons

I have written many versions of GPU kernels over the past several years and in so doing have learned many lessons about how to improve performance. The four main lessons I have learned include the following:

- Hardware support for instruction-level pipelining can be taken advantage of by increasing the work per thread via instruction-level parallelism.

- Hardware support for SIMD can be taken advantage of by partitioning data into runs and data blocks via data-level parallelism.
- Hardware support for multi-threading can be taken advantage of by increasing the number of threads launched via thread-level parallelism.
- Hardware features that can either help or hinder throughput performance.

I have generalized these performance lessons from my own GPU programming into frameworks, which I call *data access skeletons* (DASks) and *block access skeletons* (BASks). These DASks and BASks provide three main benefits:

- They support efficient access patterns into memory.
- They support experiments on instruction-level and thread-level parallelism to find the optimal balance between the two for high throughput.
- They provide a working framework that hides much of the complexity of writing GPU kernels with high performance.

In general, higher performing code is more complex, and this is certainly true for these skeletons. When writing high performance GPU algorithms, a programmer must first get a correct and working implementation up and running. So, first I introduce a simple GPU copy kernel in Chapter 4 before introducing higher performing but more complex DASk versions of Copy in Chapter 5.

Partitioning: My DASks group *n* data elements into *m* fixed-size data blocks. These *m* data blocks are then load-balanced across *p* thread blocks using different memory access patterns. There are five important benefits to this approach:

- **Parallel Processing:** By design, my DASks are built to efficiently support the 2-level *cooperative thread array* (CTA) hierarchy used for parallel processing on modern GPUs.
- **Data Coherence:** Coherent data allows efficient near-peak *input/output* (I/O) throughput into memory. My DASks support high coherence by working with blocks of data.
- **Sequential Ordering:** My Row DASk supports sequential ordering for those algorithms that require it for correct behavior (such as Scan and Radix Sort).
- **Deterministic Execution:** All my case study algorithms are implemented in a lock-free deterministic manner that does not require mutual exclusion (with one exception[7]).

[7] The one exception is the use of barrier synchronization, which makes all threads wait at the synchronization point until all threads arrive before all threads are allowed to proceed execution again. I

- **Data Independence:** Most of my case study algorithms are data independent[8].

There are three main DASks --Block, Column, and Row-- and two main BASks --Block-by-Block and Warp-By-Warp. At a high level, these all represent different access patterns (or layouts) into global memory. The GPU architecture supports parallel performance using a 2-level parallel hierarchy. The three DASks support efficient access patterns across thread blocks within a parallel grid when accessing the entire data set. The two BASks support efficient access patterns by individual threads within a single data block when accessing a single data block. These DASks and BASks are introduced and discussed in more detail in “Chapter 5 – Data Access Skeletons”.

1.3 Case Studies

Many issues become evident when working with parallelism on GPU hardware. These issues are a result of adapting serial algorithms into parallel algorithms and mapping parallel concepts onto specific GPU architectures. GPU programmers must learn how to take advantage of beneficial hardware features while mitigating harmful hardware features in order to unlock high performance. GPU programmers must learn about using parallelism (instruction-level, data-level, thread-level, bit-level) to increase performance. In order to make these abstract lessons more concrete, I implemented algorithms on GPUs. While implementing these algorithms, I of course use my DASks and BASks[9] to speed up my own development time and provide a high performing framework. Of course each case study implementation has additional valuable

use the syncthread() method for barriers within a thread block and barriers between GPU kernels happens automatically.

[8] My *k*d-tree case study algorithm is data dependent. Since each thread represents a single query point, each thread branches down its own unique path through the search tree. However, there are 32 threads per warp that move in lock-step through the code. Consequently, performance varies with data as CUDA serializes instructions from threads on different branches.

[9] One exception, my *k*d-tree case study was written before I generalized the concept of DASks and BASks and I have not yet rewritten that code on top of one of these skeletons.

lessons about performance. I showcase many of the lessons in GPU programming via five different case studies: Memory I/O via Copy, *k*d-trees, Reduce/Scan, Histogram, and Radix Sort.

1.3.1 Memory I/O via Copy:

The primary focus of my Memory I/O case study is on demonstrating all three of my DASks and both of my BASks via the Copy primitive. The Copy primitive copies *n* inputs onto *n* outputs. The secondary focus is on showing how my DASks can achieve a high percentage of peak I/O throughput on GPUs. To this end, I implement the Copy primitive in four different ways: Simple, Block, Column, and Row. Getting the simple copy kernel up and running correctly is described in more detail in Chapter 4 – Case Study Memory IO". The other more complex and higher performing versions of Copy based on my three data access skeletons (Block, Column, and Row) are described in more detail in "Chapter 5 - Data Access Skeletons". I conduct experiments on all four versions of Copy to find the best performing balance between ILP and TLP and achieve up to 30%, 82%, 77%, and 77% of peak I/O throughput, respectively.

1.3.2 *k*d-tree for Nearest Neighbor Searches:

In the *k*d-tree case study, I implement GPU kernels for nearest neighbor search[10] using a *k*d-tree[11]. With a nearest neighbor search, the goal is to find the closest point (or *k* points) within a search set of *n* points for each of *m* points in a query set. **Note:** Unlike my other case studies, this case study does not use any of my DASks.

My first exposure to GPU programming was implementing a *k*d-tree for use on nearest neighbor searches. I intended to use this nearest neighbor search as part of a terrain visualization problem on LIDAR data. My solution took much longer than I had originally budgeted. However, eventually I got it working and in time achieved a 25× speedup in performance over the

[10] Nearest neighbor searches are discussed in the following books and papers (Bentley, 1975; Bustos et al, 2006; Mount and Aray, 2010; Shakhnarovich et al, 2005).
[11] *k*d-trees are a type of spatial searching data structure created by Bentley (Bentley, 1975).

equivalent single-threaded CPU code. I was proud of this result at the time. However, looking back with the benefit of more experience, I see that my original implementation was naïve. It did not take advantage of many GPU hardware features and ran head on into several hardware limitations that constrain parallel performance. This *k*d-tree implementation is described in more detail in “Chapter 7 – Case Study *k*d-tree”.

1.3.3 Reduce/Scan:

In my Reduce/Scan case study, I use my Row DASk to implement high performance parallel Reduce and Scan GPU primitives[12]. Reduce produces a total sum by accumulating *n* inputs into a single final sum. Scan (Prefix Sum) produces a running sum by accumulating *n* inputs into *n* outputs, where the i^{th} output element is the cumulative sum of the first *i* (or *i*-1) input elements. Reduce and Scan have similar implementations. Both primitives are almost trivial (3-5 lines of code) to implement on a serial CPU. However, the parallel GPU implementations are much more complex. This complexity is a direct result of data being load-balanced across tens of thousands of threads and the requirement for partial per-thread sums to be hierarchically combined and redistributed for correct final results. The Scan primitive requires that inputs be processed in sequential order for correct scanned results. Although the Reduce primitive does not require sequential ordering, I choose to implement it the same way as Scan.

As will be seen, I perform experiments on both ILP and TLP to find the optimal balance for best performance. My Reduce and Scan primitive achieve up to 89% and 85% of peak I/O throughput on the GTX 580 (and up to 76% of peak I/O throughput on the GTX Titan). My Reduce/Scan primitives are described in further detail in Chapter 6.

[12] For more details about the Reduce and Scan primitives, see the following papers (Blelloch, 1989 and 1990; Blelloch and Maggs, 1996; Chatterjee, 1990; Harris et al, 2008; Hillis and Steel, 1986; Merrill and Grimshaw, 2010 Parallel Scan).

1.3.4 Histogram:

In my Histogram case study, I use my Column DASk to implement a parallel 8-bit histogram primitive on the GPU. A histogram[13] summarizes the frequency distribution of an entire data set via a much smaller table of counts. In a nutshell, *n* input elements are counted into *m* bins. The resulting *m* frequency counts form the histogram output. Each of the *n* inputs is assumed to be taken from a range, R = [min, max). Each of the *m* bins represents a sub-range r_i of R. (These sub-ranges uniquely partition and fully cover the original range R). Counting proceeds by selecting the matching sub-range for each input element and incrementing that bins counter.

An 8-bit histogram can be implemented using a simple indexing operation on 8-bit data. The serial CPU implementation is trivial (5-8 lines of code). Although histograms are straightforward to implement on a sequential CPU, they have proven difficult to adapt for use on GPUs with low performance results in prior GPU histogram implementations. I ran into similar performance issues since my GPU Histogram only achieves up to 21% of Peak throughput on the GTX 580. Nevertheless, my GPU Histogram still runs up to 50% faster than prior GPU histogram methods for random data and up to 2-4× faster for image data. My 8-bit GPU histogram is described in further detail in Chapter 8.

1.3.5 Radix Sort:

In my Radix Sort case study, I use my Row DASk to implement a parallel Radix Sort on the GPU. Even though a serial radix sort[14] is straightforward to implement (as a 3-step Counting Sort pass over each *r*-bit digit within a numeric key), the corresponding CPU/GPU hybrid radix sort is much more complex. This complexity arises due to the need to load-balance data across tens of thousands of threads, hierarchically scan counts into starts, compress/decompress data,

[13] Histograms were created by Pearson (Pearson, 1895).
[14] Radix Sort algorithms are discussed in the following books (Cormen et al, 2009; Sedgewick, 1998).

and many other complex actions required to overcome hardware limitations. My hybrid solution has the CPU implement the radix sort as multiple passes over 4-bit digits within each 32-bit numeric key and then invoke three GPU kernels (`GPU_CountKeys`, `GPU_ScanCounts`, and `GPU_DistributeKeys`) to do a full counting sort on each chosen digit in each pass.

As will be seen, I perform experiments on both ILP and TLP to find the optimal balance for best performance. My GPU radix sort can sort up to 717 and 836 million ‹*key*/*value*› pairs per second on the GTX 580 and GTX Titan respectively, which I estimate[15] are about (59% and 46%) of peak data throughput rates respectively. My Radix Sort is described in further detail in Chapter 9.

[15] Given 32-bit keys and 32-bit values with a 4-bit digit, the radix sort requires 8 passes (=32/4) to fully sort the data. Given these assumptions, I estimate that the maximum data throughput rate is 1205 million and 1802 million ‹*key*/*value*› pairs per second on the GTX 580 and GTX Titan respectively.

1.4 Summary:

As will be demonstrated in my various case studies, my DASks and BASks help programmers take advantage of the massive amounts of parallel processing power available on modern GPUs. Even though writing GPU data parallel code that is both correct and high performance is quite difficult, my three DASks help solve many of the issues that GPU programmers must grapple with and eases the burden of implementing their own GPU programs. These DASks use generic programming techniques to enable experiments on both ILP and TLP techniques, in order to find the optimal balance between the two for best performance.

2.0 Parallelism

As mentioned previously, graphics processing units (GPUs)[1] achieve high levels of performance mainly through their massive use of parallelism. A thorough knowledge of the types of parallelism available is vital for anyone programming for a GPU. I therefore in this chapter review the common types of parallelism used by GPUs—Flynns taxonomy, machine models, and memory models—as well as go over the the Fermi and Kepler architectures, which are particularly helpful in my case studies.

2.1 Types of Parallelism

Parallelism is the simultaneous processing of several tasks or multiple data items on multiple hardware processing units. These units are called *cores*[2] or, in recent parlance, *multi-cores* to distinguish them from the simple *single-core* CPU[3] architectures of older computers. Each processing core is assumed to work on its own independent instruction stream to perform useful computations to accomplish a task. Typically each task involves transforming input data streams into output result streams.

Parallel processing[4] often requires communication and coordination between cores to accomplish the original task. *Granularity* is a qualitative measure of the amount of computations done compared to the communications done. *Coarse-grained parallelism* implies that large amounts of computations are done between communication / coordination events. *Fine-grained*

[1] Recall that GPU stands for *graphics processing unit*, GPUs are massively parallel processing machines.
[2] Also known as *processors*.
[3] Recall that CPU stands for *central processing unit*, CPUs are thought of as serial processing machines even though modern CPUs are actually multi-threaded multi-core devices that typically support multiple forms of parallelism.
[4] As described in the book Computer Architecture (Hennessey and Patterson, 2012).

parallelism implies that small amounts of computations are done between communication / coordination events. *Parallel overhead* is the total amount of time required to communicate and coordinate work between parallel tasks, as opposed to doing useful work solving the original problem.

2.1.1 Flynn's Taxonomy

Flynn's taxonomy (Flynn 1972) offers a popular high-level way of categorizing the different types of parallelism and shows the fundamental differences between CPU and GPU architectures. In fact, by imagining an evolution of a CPU core into a GPU core, we will see differences in machine models, memory hierarchies, and the way to think about mapping parallel computation onto the underlying architecture.

Flynn's Taxonomy groups serial and parallel models into 4 broad groups, either single or multiple along two axes (instructions and data): *Single-Instruction Single-Data* (SISD), *Multi-Instruction Single-Data* (MISD), *Multi-Instruction Multi-Data* (MIMD) and *Single-Instruction Multi-Data* (SIMD). These groups are useful in categorizing parallel hardware, software, and models. For example, Figure 2.1 shows the four types of parallelism used in my case studies: instruction-level parallelism (ILP), thread-level parallelism (TLP), task-level parallelism, and data-level parallelism (DLP).

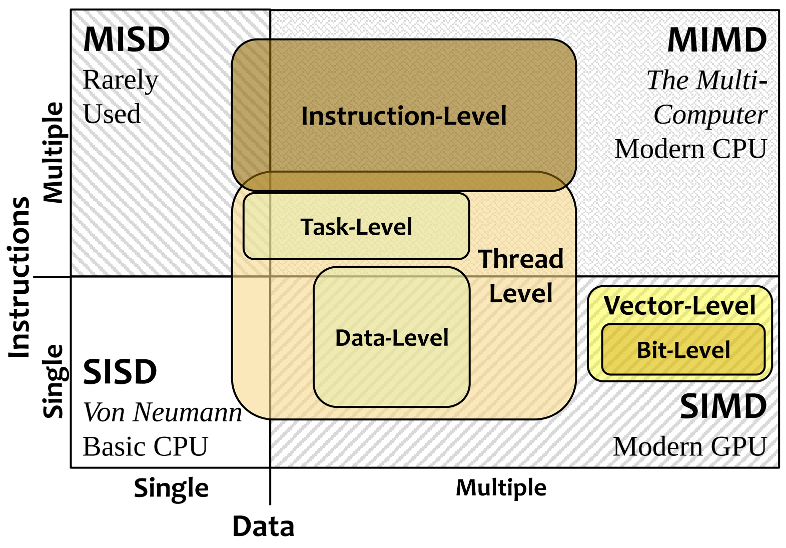


Figure 2.1: I categorize Instruction, Thread, Task, and Data-level Parallelism using *Flynn's Taxonomy* (SISD, MISD, MIMD, and SIMD). (Less importantly, I also categorize Vector- and Bit-level parallelism.)

Instruction-Level parallelism (ILP) executes instructions on multiple processing cores to increase the instruction throughput of a serial instruction stream. On CPUs, ILP has a huge effect on the underlying architecture, but thanks to the heroic efforts of hardware architects and compiler writers, most programmers see little effect on the common programming model, and can continue to think of and work with modern CPUs as if they were simple von Neumann SISD computers instead of the complex MIMD *systems on a chip* (SoC) that they have become.

Thread-level parallelism (TLP), also known as *multi-threading*, maps multiple streams of execution onto multiple cores. Initially, multi-threading appears to be clearly in the MIMD category for both the architecture and the programming model. In fact, some of the first uses of TLP was to hide latency on SISD CPU architectures by switching from stalled threads to other active threads to continue doing useful work. On SIMD processors, batches of threads (known as *warps* on NVIDIA GPUs) are mapped onto batches of simple processing cores.

TLP enables task-level and data-level parallelism. *Task-level parallelism* divides one large task into several smaller subtasks and maps each subtask onto its own thread or warp. *Data-level parallelism* (DLP) partitions a large data set into smaller runs of data and also maps each run onto its own thread or warp. I also show two other types of parallelism (bit-level parallelism (BLP) and vector-level parallelism (VP) that will I will revisit in my case study on Histograms (chapter 8).

See the book *Computer Architecture*[5] (Hennessey and Patterson, 2012), for a more in-depth description of each type of parallelism above (as well as other types of parallelism).

2.1.2 Machine Models, Memory Models, and the Memory Hierarchy

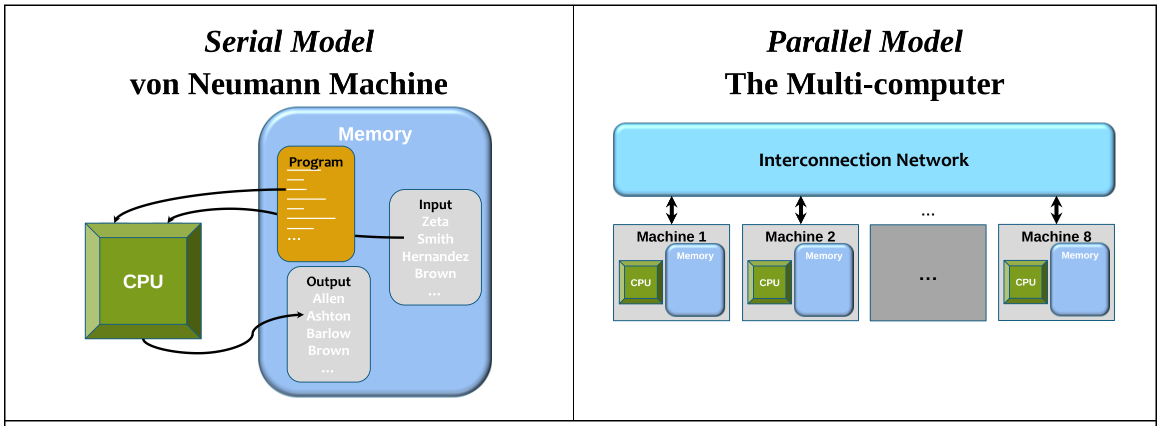


Figure 2.2: Serial machine model vs. Parallel machine model. The classic serial *von Neumann model* abstracts a sequential CPU plus a load/store memory that contains both instructions and data. *The Multi-computer model* abstracts parallelism by connecting multiple serial computers together via a network and adding support for remote message passing and remote memory access.

Flynn's taxonomy includes data and instruction streams but does not cover types of memory access. Yet as we will see in my case studies, useful models of memory access are also important to GPU programmers in order to achieve take advantage of the performance characteristics and features of the GPU memory hardware. Two machine models for thinking about instruction, data, and access parallelism are the von Neumann computer (von Neumann, 1945) and the multicomputer, shown in Figure 2.2. *Machine models* abstract hardware for programmers so that they can focus on high-level algorithms, data-structures, and implementations without getting mired in the technical details of each machine's specific architecture.

[5] For more information about different types of parallelism, see the book *Computer Architecture: A Quantative Approach, 5th Edition* by Hennessey and Patterson.

The original *machine model* is the *von Neumann computer*, which consists of one CPU performing serial computations. The CPU connects to a storage unit called *memory*. All programs and data are stored in memory. Instructions and data are transferred between CPU registers and memory using load/store operations. The CPU executes a program containing a sequence of instructions to transform input data into output results.

The *multi-computer* contains a number of simple von Neumann computers called *nodes* that are linked to each other over an interconnection network. Each node executes its own program. Each program may access its own local memory directly. Each node may communicate and coordinate with other nodes over the network by sending and receiving messages. These simple abstract models have performed well for over sixty years by allowing advances in software to proceed independently from advances in hardware.

Under Flynn's Taxonomy, a von Neumann machine is SISD, and the multicomputer is MIMD. For serial programming, programmers follow the *random access memory* (RAM) model and assume memory access costs are fixed regardless of the machine's physical location.

For parallel programming, there are two main memory models: distributed memory and shared memory[6]. For distributed memory, programmers explicitly use remote message passing over the interconnect network to communicate and coordinate parallel CPU nodes. In the ideal distributed model, message costs are proportional to message length. However, in practical models, message costs are impacted by three factors: 1) the topological layout of the network (meaning, cube, star, ring, etc.), 2) the physical distance between nodes, and 3) the number of messages competing for concurrent use of the interconnect network.

For the shared memory model, a large shared memory pool replaces the interconnect network. This model supports a concurrent RAM model. With it, every node has a direct

[6] *Shared memory* is a general term in the parallel programming community for accessing memory in parallel across multiple cores. Unfortunately, NVIDIA also uses the same phrase to indicate a specific type of memory on GPU cores. This may result in some confusion for the reader.

connection to the shared memory pool that all nodes share. Each node also has direct access to its own attached local memory pool. Because of communication overhead, access to other nodes shared memory is slower than direct access to its own local memory. The number of nodes involved in this model tends to be small, often 2 to 8, because the all-to-all communication required to support concurrent coherent access to shared memory grows geometrically as node are added.

In this shared memory context, coherent access implies that all changes to memory by any node are visible to all other nodes in the multicomputer. Programmers can implicitly use shared memory to communicate data and coordinate execution across parallel nodes. The concurrent RAM model is easy for most programmers to understand since it is most similar to the RAM model they are already familiar with. However, concurrency implies that multiple nodes can update the same memory locations at the same time leading to resource competition. Programmers must prevent or manage resource competition between nodes to avoid serious problems, such as incorrect results and crashes.

Computer architects design and build memory architectures that are far more complex than the simple abstract models (RAM and concurrent RAM) programmers use to understand them. Because different memory technologies have many orders of magnitude differences in how they make trade-offs between cost, speed, and capacity, architects build large memory hierarchies, both architects and compiler writers try to hide the hierarchy and simulate the simpler memory models for the average programmer. Because, in my case studies, the memory hierarchy of the GPU has a large effect on the techniques to improve performance, I want to review some of the terminology and issues in modern memory architectures.

Architects divide the memory system into multiple levels (Hennessey and Patterson, 2012), with each level containing its own memory type with its own unique characteristics—the primary focus is on speed, size, and cost. The entire memory system achieves high transfer speeds by exploiting locality (both spatial and temporal). If an algorithm asks for a variable stored in a slower memory level, move and store that variable into a faster memory level, a process called *caching*. Caching is effective because if an algorithm just accessed a memory location, that algorithm is likely to access the same location again soon, a concept called *temporal locality*. While the memory system is transferring data, it should also go ahead and grab a small fixed size neighborhood of memory around the requested memory location and cache that as well. Because if an algorithm just accessed a memory location, then that algorithm is likely to access that location's neighbors as well, a concept called *spatial locality*. Frequently accessed memory ends up being cached in higher speed memory closer to the CPU resulting in better I/O throughput. In this locality context, *coherent memory accesses* mean repeated memory accesses clump up close to each other either in time and/or space. Coherent memory accesses run close to the speed of the fastest memory layer involved. While random or incoherent memory accesses run close to the speed of the slowest memory layer involved. This is because coherent memory accesses are well localized (across time or space), which caching exploits.

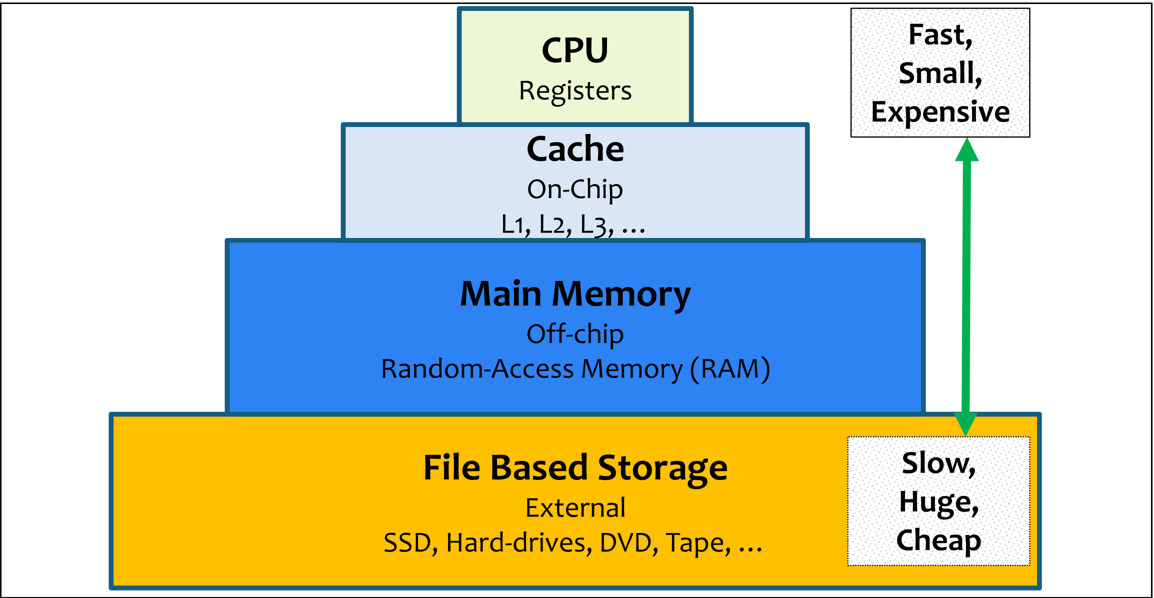


Figure 2.3 Memory Hierarchy with 4 broad tiers – very fast CPU registers, fast on-chip cache, medium speed off-chip RAM, and slow external file-based storage. Memory is typically sorted by access speed with fast memory kept close to the CPU and slow memory kept more distant. Architects use a memory hierarchy, caching, and the principle of locality to support fast average access speeds at a reasonable total cost.

Architects design modern computer memory in a hierarchical manner, as shown in Figure 2.3, to provide fast memory access at low cost. A typical memory system consists of four broad layers. The CPU layer consists of registers used to store input operands and output results. This memory is very fast but very expensive so designers keep it very small. The on-chip multi-level cache layer exploits locality by temporarily storing larger fixed size chunks of instructions and data. This memory is fast but expensive so it is kept small. The off-chip main memory layer consists of large amounts of random access memory (RAM) shared across all cores on the chip. This memory is reasonably fast, and affordable so it is kept large [1-32 GB]. Finally, the storage layer typically consists of file-system based hardware stored externally in devices such as USB keys, SSDs, DVDs, and hard drives. This memory is slow but cheap so designers can afford huge amounts of storage.

In theory, the RAM model treats memory access as having a fixed cost to make it easier to develop new algorithms. CPU architects and compiler writers put in great efforts to provide an environment that most programmers can see as following the RAM model (at least if they respect some heuristics about accessing data with good locality, and partitioning drives wisely). However, serial CPU programmers for whom performance is crucial may need to know much more about the memory hierarchy and may even circumvent some of the architects' efforts by memory mapping or by using other tricks that operate on lower levels of abstraction. Currently, GPUs have less architecture and compiler support for the concurrent RAM model. So, in practice, parallel GPU programmers have to much more aware of the memory hierarchy, memory types, and varying costs for accessing memory at each level.

2.1.3 Deriving a GPU core from a CPU core

The following diagrams shows the broad differences between CPU and GPU architectures, it summarizes a PPAM[7] tutorial by Göddeke (Göddeke et al, 2011). It shows how to evolve a CPU into a GPU in four steps: *simplify*, *replicate*, *re-organize*, and *warp-thread*.. I place these in their categories in Flynn's taxonomy in Figure 2.4.

At a high level, CPUs contain a few (2-8) large and complex ILP focused cores that favor coarse-grained Task-Level parallelism, whereas GPUs contain hundreds of simple TLP focused cores that favor fine-grained data-Level parallelism. This differentiation can be seen as arising naturally from the following steps.

[7] This summary is based on a *Parallel Processing and Applied Mathematics* (PPAM 2011) tutorial session on "Scientific Computing on GPUs" by Göddeke et al.

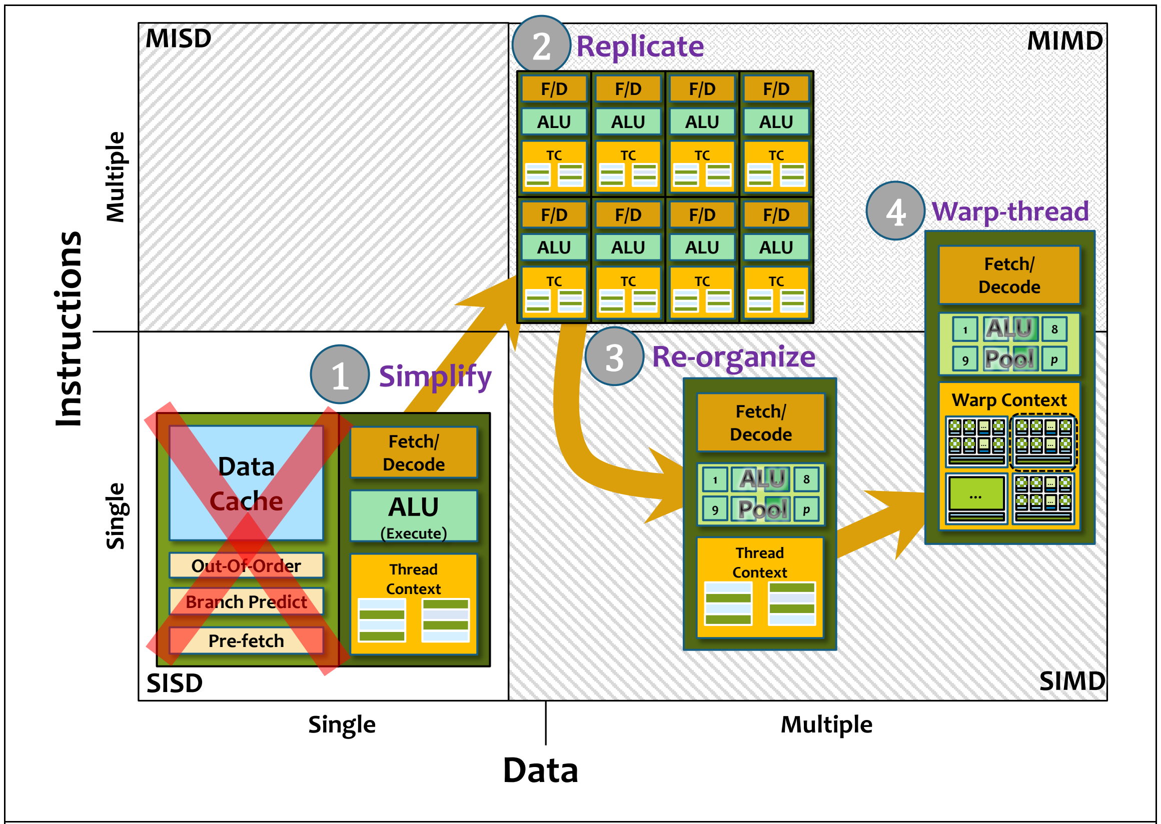


Figure 2.4: Conceptual differences between a CPU core and a GPU core in broad themes. Based on a talk by Göddeke (Göddeke et al, 2011). Start with a complex CPU ILP-focused core. 1) *Simplify:* Remove complex ILP and large data caches to get a simple processing core. 2) *Replicate:* Repeat that simple core many times on the GPU chip to support massive parallelism. 3) *Re-organize:* Combine control logic for many simple cores into one larger single vectorized SIMD core. 4) *Warp-thread:* Support multi-threading at a group level, provide large context pools and hardware support for rapid context switching between groups of threads called warps. Finally, arrive at a GPU TLP-focused core.

Simplify: The CPU is conceptually reduced to its essential functions for executing programs, keeping the fetch/decode control logic, an arithmetic logic unit (ALU) for computation, and some execution context pools to support multi-threading. What is eliminated to save transistors is the large on-chip caches and most of the complex logic used to implement instruction-level parallelism (ILP) techniques. These simple cores will of course run slower than modern CPUs as they do not exploit serial parallelism or memory locality.

Replicate: The saved transistors are then spent on creating many parallel simple cores on a single chip. To support parallel processing properly, control logic to schedule multiple threads onto multiple cores would also need to be added. This MIMD approach can support both task-level and data-level parallelism. Algorithms that divide tasks or data across multiple cores will see a large performance gain due to parallelism.

Reorganize: The core architecture is redesigned by reorganizing all of the ALU cores and their execution contexts into larger resource pools. To simplify yet further, a single instance of the instruction, control, and scheduling logic is shared across all the pooled ALU cores. This reduces the total transistor count at the cost of requiring all simple cores to move in lock-step through the same instruction stream. This approach is similar to vector-parallel computing and is at the heart of SIMD processing. For now, I will call this larger pooled core a *SIMD core*. This approach reduces system overhead by amortizing the cost of managing the instruction stream across many simple cores. Each simple core within the larger SIMD core works on the same instruction but on different data values, which strongly favors data-parallel algorithms.

Warp-thread: Support is added for a large number of execution contexts for multi-threading in batches of threads. Using NVIDIA's terminology, this batch of threads is known as a *warp*. Warp-threading allows the GPU to store a large number of dynamic warps and the associated state (or context) for each individual thread within each warp and additional context at the warp level. Each warp context corresponds to a batch of contexts (one per simple core) on the SIMD core. Think of warp-threading as multi-threading that occurs at the SIMD core level, not at the ALU core level with each warp-thread acting as a fat bundle of smaller threads (one per ALU core) that all move in lock-step through the instruction stream. All simple threads within a warp thread share a single warp context, including a program counter and other related resources (such as instruction cache or assigned shared memory) to march through the same instruction stream as a group.

Next, imagine adding support for a fast fine-grained multi-threaded (multi-warp) scheduler to rapidly switch between warp contexts as individual warps stall during execution. NVIDIA calls this thread manager a *warp scheduler*. Individual warps may stall, however, if this happens, the warp scheduler will rapidly switch the SIMD core to another active warp and keep all the simple cores within each SIMD core doing useful work. In this way, the warp scheduler keeps the SIMD core busy by hiding latency and increasing overall instruction throughput.

After simplifying, replicating, re-organizing, and warp-threading, we arrive at a convenient abstraction of a modern GPU architecture.

2.2 GPU Architecture

A modern GPU core, as found on the GTX 580, as shown in Figure 2.5, is a result of combining the broad concepts from section 2.1.3 into silicon. In NVIDIA terminology, A replicated, re-organized, and warp-threaded SIMD core is called a *streaming multi-processor* (SM). Each SM contains multiple *scalar processors* (SP) similar in concept to simple ALU processing cores. Warp-threading is supported using warps of 32 threads that in turn are mapped onto the physical SPs. The instruction (fetch/decode), control and warp scheduling logic are shared across all the SPs on the SM.

The resulting GPU architecture supports data-level parallelism via both ILP and TLP concepts. ILP is supported in hardware by *pipelining* and *scoreboarding* on each SM core. TLP is supported by warp level multi-threading across lots of SP cores. GPUs take the *replicate* concept one step further by replicating each SM multi-core across the chip multiple times to achieve the massively parallel computing devices known as GPUs.

Simple Core	**SIMD Core**	**Modern GPU**
SP Core (*Scalar Processor*)	SM Core (*Streaming Multi-Processor*)	GPU Card (GTX 580 = Fermi)

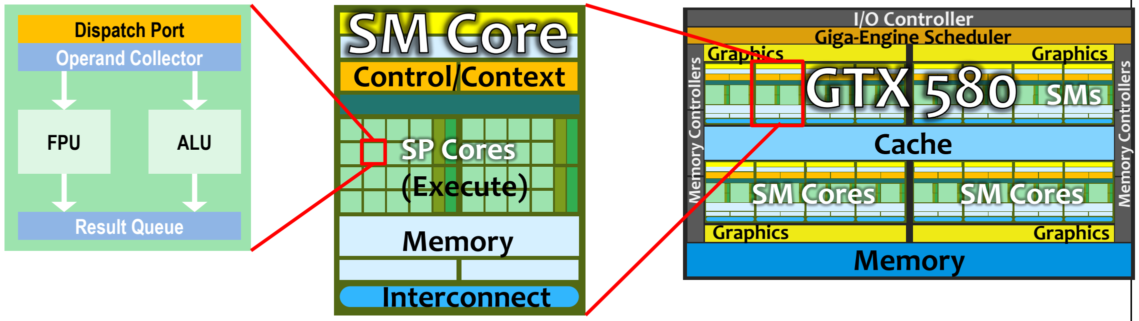


Figure 2.5: A modern GPU core: The Fermi multi-processor core implements massive parallelism by simplifying, replicating, re-organizing, and multi-threading. On the left, is a simple pipelined SP-core (*scalar processor*). In the center, 32 SP cores are replicated and then re-organized with shared instruction, control, and scheduling logic into a SIMD core called a SM-core (*Streaming Multi-processor*). The on-core *warp scheduler* supports rapid context switches between up to 48 different warps, containing 32 parallel threads each, for a total of up to 1,536 threads in 48 independent warps of execution. NVIDIA replicates again putting 16 SM cores onto each GTX 580 GPU card for a total of 512 simple cores per card. NVIDIA also adds support for an intra-core thread scheduler, 3D graphics, a memory hierarchy, 6 memory controllers and an I/O controller.

So, in a broad sense, CPUs spend silicon on advanced ILP techniques and large data caches to speed up the performance of serial algorithms. GPUs spend silicon on warp-based multi-threading and massive numbers of simple cores to speed up the performance of data-parallel algorithms.

Hardware architects and compiler designers have done an excellent job hiding the actual parallel complexity of modern CPUs from programmers. Most programmers can safely assume a RAM memory model on top of a von Neumann machine and achieve solid results. However, GPU chips and the CUDA[8] platform (NVIDIA, 2010, What is CUDA) are not as evolved and programmers must put in greater efforts to understand the underlying GPU architecture and issues entailed in order to make their kernels run correctly, robustly, and with high performance. To help with that deeper understanding, I will drill down on the GPU architecture in greater detail in this section.

[8] CUDA is an abbreviation that stands for Compute Unified Device Architecture as defined by NVIDIA.

A GPU supports high-performance parallel computing by combining together these three elements:

1. A hybrid (MIMD/SIMD) hardware model for data-level parallelism
2. A massive multi-threaded two-level scheduler for thread-level parallelism
3. Simple programming language extensions for ease of parallel programming

GPUs support incredible processing rates for massive data sets that can be adapted to their models of processing, memory, and communications. This adaptation is not easy, because these are complex, interacting models, with unexpected limitations imposed by hardware. I suggest that, because the hardware architects have focused on implementing a data-parallel processor model, have given flexibility for the memory model, and have spent less effort on the communication model, the programmer may wish to consider these in reverse order: first making algorithm selections based on communication constraints, secondly choosing parameters based on memory hierarchies, and then implementing kernels using parallel thread groups (or warps).

Now, that we have a high-level understanding of how GPU's differ from CPU's, I will next explore the GPU architecture from several different points of view:

1. Processing and memory,
2. Warp-threading and scheduling
3. Communication and coordination

Any parallel computation devices must provide mechanisms for instruction dispatching and processing, memory storage and retrieval, thread scheduling and management, and communication/synchronization. GPUs were designed for fast processing in graphics pipelines, in which the same computation (viewing transformation, illumination, z-buffering, texture-mapping) can be performed in parallel on a large amount of data. Thus, the GPU hardware architects have created a fixed hierarchy of processors, a flexible hierarchy of memory, two levels of multi-threaded schedulers, and a limited set of primitives for communication / coordination across threads. I will overview the NVIDIA Fermi GTX 580 card and Kepler GTX 680 cards as specific examples of these high-level design concepts.

2.2.1 Hardware Processor and Memory Hierarchies

Current GPU architectures use a two-level hierarchy of processing cores. Consider, as examples, NVIDIA's Fermi and Kepler architectures, The GTX 580 GPU (NVIDA, 2010, Fermi) contains 16 SMs, each containing 32 SPs for a total of 512 cores with an aggregate 1.58 Tera-FLOPs of single-precision peak compute capacity. The GTX 680 (NVIDIA, 2012, GTX 680) contains 8 SMXs, each with 192 SPs for a total of 1,536 cores with an aggregate 3.10 Tera-FLOPs of single-precision peak compute throughput. On both the GTX 580 and GTX 680, double-precision peak throughput drops to only 197 and 129 Giga-FLOPS, an 8-fold and 24-fold decrease, respectively, instead of the expected two-fold, because only a few of the FPUs can handle 64-bit doubles.

Both processors execute hierarchically organized threads, which are best considered as single-instruction/multiple-data (SIMD) processes, as we will describe shortly in section 2.2.3.

Memory Types: Each GPU has support for many different types of memory (NVIDIA, 2012 Programming Guide) --*Registers*, *Shared*, *Global Memory*, *Cache*, *Surface*, *Texture*, *Constant*, and *Local*, as shown in Table 2.1. The Surface, Texture, and Local memory are just special forms of global memory with extra functionality and/or behavior. From a performance point of view, Registers are the fastest form of memory, followed by Shared memory (L1 Cache), L2 Cache, and finally Global Memory. There is also even slower access to Host CPU RAM via DMA transfer. I will briefly overview each type of memory in this section.

Registers	**Shared Memory** (L1 Cache)	**Global Memory** (L2 Cache)
• 32K registers per SM • Each register is 32-bit (4 bytes) • 128KB per SM • 2048KB aggregate total • At least 8 TB/s peak aggregate compute throughput	• 48KB shared • 16K L1 cache • or (16K/48K) • 32 banks (4 bytes per bank) • Bank conflicts • 1.43 TB/s peak aggregate I/O throughput	• 1.5 GB capacity • 6 memory controllers • 768 KB L2 cache shared across all SM's • 4-8 GB/s data transfer GPU ↔ CPU • 192.4 GB/s peak aggregate I/O throughput.
Constant Memory (CUDA API)	**Texture Memory** (CUDA API)	**Local Memory**
• Read-only memory • 64 KB • 2KB cache per SM • Use Broadcast mode otherwise access is serialized. • Can be as fast as registers. • Declare variables/arrays as constant to use.	• Intended to support textures for 2D and 3D graphics • Read-only memory • Must be properly initialized before using. Created out of global memory. • Has own separate texture cache • Supports a variety of pixel formats. • Supports filtering/interpolation • Supports addressing operations (clamping, tiling, mirroring)	• Not really a memory type but a platform behavior to deal with *Register Spill.* • Local variables that exceed the number of assigned registers per-thread are stored in global memory. • These variables are said to be in *Local* memory. • These variables run at global memory speeds impacting performance.

Table 2.1 - GPU Memory Types: High level summary of the main GPU memory types.

Registers: Each SM (or SMX) has a large pool of registers. The registers can be flexibly assigned to individual threads. The registers (4-bytes each) can be partitioned and directly assigned across all concurrent threads scheduled on each SM. On the GTX 580, each SM has 32K of 32-bit registers with an estimated peak aggregate throughput of up to ~9.5 TB/s for the *Fused Multiply Add* (FMA) instruction. On the GTX 680, each SMX has 64K of 32-bit registers with an estimated peak aggregate throughput of up to ~18.5 TB/s for the FMA instruction. Most other ISA[9] instructions tend to run at about half of these peak FMA rates, so ~4.7 TB/s and ~9.2 TB/s respectively is more typical.

Shared Memory: Each SM (or SMX) has another pool of local memory. The pool of local memory, currently 64KB on both the GTX 580 and 680, can be split between two

[9] Recall that ISA stands for *instruction set architecture*, i.e. the various processing operations (arithmetic, logic, shifts, conditionals) that a specific chip architecture supports.

categories: L1 cache and shared memory. The local memory split can be either 16/48, 48/16 or 32/32 KB (the last grouping on Kepler class cards only) (NVIDIA, 2012, Kepler GK110). The L1 cache memory speeds up read-only access to global memory via temporal and spatial locality. The cache-line size, also known as a *warp-line*, is 128 bytes (or 32 32-bit elements). This local memory is called *shared memory* by NVIDIA since it is shared by all SPs on each SM (or SMX).

Shared memory is available as a programmable scratch pad to store local variables and arrays. This scratch pad memory allows limited communication and coordination across threads and can help speed-up overall performance. The programmer has direct control over how much memory to request out of the shared memory on each SM for each thread block. However, up to 8 (or 16) concurrent thread blocks per SM (or SMX) need to share the same pool of shared memory. The programmer designates the amount of shared memory as local arrays assigned to each thread block. The CUDA platform figures out how many concurrent thread blocks can run at the same time based on the amount of memory the programmer requested and the amount of shared memory available. CUDA then partitions the shared memory across the concurrent thread blocks per SM. For example: If the programmer declared a memory array of 2,000 32-bit elements in shared memory, this would require 8,000 bytes. CUDA would decide that it could run at most 6 concurrent blocks ($6 = \lceil 49{,}152/8{,}000 \rceil$ bytes, where 49,152 = 48KB). CUDA could decide to layout the 6 blocks of shared memory at starting local offsets of $[0;\ 8{,}000;\ 16{,}000;\ 24{,}000;\ 32{,}000;$ and $40{,}000]$ respectively.

To increase throughput, shared memory is divided into 32 memory banks with a width of 4 bytes per bank. The GTX 680 also supports another addressing mode with a width of 8 bytes per bank. To prevent contention, if more than one concurrent thread accesses the same bank at the same time, then there is a *bank conflict*, and the hardware stalls threads on the SM (SMX) to serialize access and enforce correct behavior. On the GTX 580 and GTX 680, the peak aggregate throughput of shared memory is 1.58 TB/s and 1.03 TB/s respectively.

Global Memory: All processors have access to a large *global memory* on the GPU card. The GTX 580 and GTX 680 have total memory of 1.5 Gigabytes and 2.0 Gigabytes respectively. (Thus, this is really "shared memory," as that term is normally used in parallel programming circles, see chapter 2.1.2, although not in GPU terminology.) Scatter/Gather operations are supported on each SM (SMX) between registers and global memory. On the GTX 580, six memory controllers manage 768KB of read/write L2 cache that is shared among all SMs and can provide an aggregate 192.4 GB/s of peak memory I/O throughput. On each SM, there is also 16KB (or 48 KB) of read-only L1 cache that is banked in the same manner as shared memory. On the GTX 680, four memory controllers manage 512KB of read/write L2 cache that is shared among all SMXs and can provide an aggregate 192.2 GB/s of peak memory I/O. On each SMX, the read-only L1 cache is typically 16 KB (but can also be setup as 48 or 32 KB).

There are three other types of memory found on GPU cards, which I briefly describe: constant, texture, and local memory. Other than these brief overviews, I intend not to discuss these three specialized memory types further in my thesis.

Constant Memory: *Constant* memory is a small read-only memory with its own small cache that can be used to store constant variables, which can be declared at compile time or via the CUDA API. Access to these constants must be done in broadcast mode (IE all threads within a warp access the exact same constant address at the same time) otherwise access is serialized impacting performance.

Texture Memory: *Texture* memory is intended for use in the 3D graphics rendering pipeline. Texture memory is read-only and supports many pixel formats, plus a lot of special functionality including addressing modes, filtering/interpolation, etc. that can be applied to pixel-data when reading from texture memory. Since the silicon for this extra functionality has already been built into the chip to support 3D graphics, using this extra functionality for compute purposes is effectively free. Even though textures are actually stored in global memory, texturing uses its own separate cache, independent of the L1 cache used for compute operations reading from global memory. Access to texture memory is via a CUDA specific API.

Local memory is not really a memory type but a platform behavior to deal with *register spill*. *Register spill* is when the number of local variables assigned to a thread by the compiler exceeds the number of physically available hardware registers. NVIDIA's solution is to store the over-flow variables in global memory. Read/write access to these *"Local"* variables runs at global memory speeds impacting performance. The L1 cache can help defray the access cost but only for read-only variables.

Memory Hierarchy: Similar to a modern CPU memory architecture, the GPU memory architecture is arranged into a complex hierarchy of Registers, L1 Cache, Shared Memory, L2 Cache, GPU RAM, and CPU RAM. CPU cache memory is typically hidden from the CPU programmer. However, GPU shared memory is accessible to the GPU programmer and can be used for caching frequently re-used data or to communicate data or coordinate behavior across the threads within a thread block. The peak throughput estimates (Volkov, 2010) above for the 3 main types of memory (registers, shared, global) suggest that on the GTX 580 using registers can be up to 6.0× faster than shared memory, which in turn can be up to 8.2× faster than global memory. On the GTX 680, registers can be up to 6.0× faster than shared memory, which in turn can be up to 5.4× faster than global memory. Based on these results, programmers should favor performing computations in registers over shared memory and performing computations in shared memory over global memory. Memory transfers between CPU and GPU memory run at a slower 4-8 GB/s peak throughput as compared to the ~192 GB/s peak throughput of global memory. Programmers should minimize transfers between the CPU and GPU as a result.

2.2.2 Warp-Threading and Scheduling

GPU data-parallel programming is accomplished using one of several API platforms: OpenCL (Khronos Group, 2012 OpenCL), Thrust (NVIDIA, 2012 Thrust; Bell and Hoberock, 2012), Microsoft's C++ AMP (Microsoft, 2012 C++ AMP), and NVIDIA's CUDA (NVIDIA 2012, What is CUDA). These API platforms unlock access to the massive parallelism available on modern GPU hardware. All current API's support the C++ language and typically enhance their respective C++ platforms by adding new syntax, keywords, data types, libraries, and API functions. These platforms also provide a compiler, linker and dynamic loader that allow GPU parallel kernels to be compiled, linked, loaded, and invoked from a CPU host.

CUDA: Since, I have only passing familiarity with the other platforms; I will focus on CUDA (NVIDIA 2012, Best Practices) in the following discussion. CUDA (formerly called the *Compute Unified Device Architecture*) is a high level language and parallel computing platform created at NVIDIA. The language is based on C++ and version 5.0 of CUDA uses the popular Low Level Virtual Machine (LLVM) compiler infrastructure to compile, link, and generate run-time GPU code from CUDA kernel programs. The CUDA language includes several C++ extensions to express parallelism, data locality, memory usage, and to manage thousands of threads. The two-level thread schedulers work with *grids* of *thread blocks*, called *co-operative thread arrays* (CTA) by NVIDIA. Thread blocks are scheduled onto individual SMs by the top-level *Giga-engine* scheduler. Concurrent warps of threads from thread blocks on each SM are managed by the *warp scheduler* built into each SM.

The LLVM compiler driver (nvcc.exe) translates the high level CUDA C++ into a lower level PTX (*Parallel Thread Execution*) assembly code like intermediate language meant to be consumed by a virtual machine. PTX is a stable *Instruction Set Architecture* (ISA) intended to span multiple GPU hardware generations. PTX is a machine independent ISA for language compilers to target. The PTX ISA has high-level support for integer arithmetic, floating point arithmetic, comparisons, Boolean arithmetic, data movement, data conversion, video, textures, surfaces, parallel synchronization, parallel communication, and control flow. A PTX optimizing assembler (ptxas.exe) converts the PTX into actual machine code for the target GPU device using a *Just-in-Time* (JIT) virtual machine. PTX supports predicated execution of any instruction in the ISA.

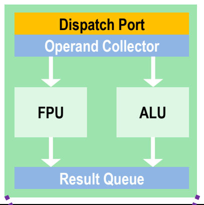
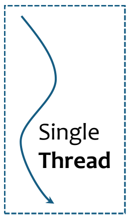
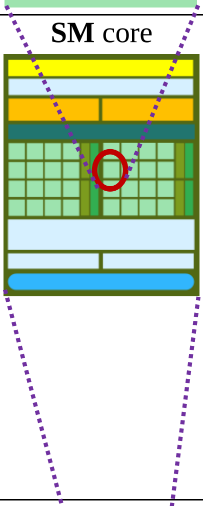
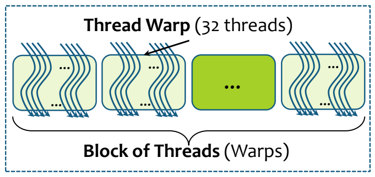
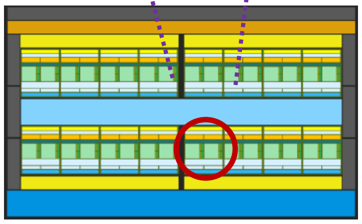
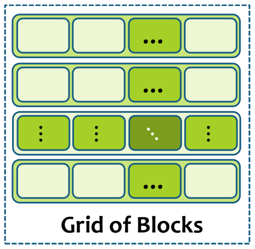
Hardware	Threading	Scheduling
SP core  Dispatch Port Operand Collector FPU ALU Result Queue		**SP:** Threads are executed by SPs. • Thread state is maintained in assigned registers. • Threads only exist as part of a larger warp and block.
SM core 		**SM:** One *kernel* per SM at a time. • SIMD execution across all SP's on SM • *Kernels* are executed via thread Blocks • *Blocks* are mapped onto SM's • Up to 8 (16) concurrent blocks per SM. • Up to 48 (64) active warps per SM. • Fine-grained scheduling. • Can switch as often as once per cycle From stalled to active warps. • SM schedules executing warps from pool of active warps. • Execution is one *Warp* per scheduler • Barrier Synchronization within *blocks* is supported
GPU 		**GPU:** A *kernel* is launched onto a CTA (*Grid* of *Blocks*) • Coarse grained scheduling • GPU waits until current set of blocks are finished before scheduling next set of blocks onto SMs. • Communication via *shared* or *global* memory • Coordination via atomics, barrier synchronization, or *kernel* termination.

Table 2.2 - GPU Threading and Scheduling Model: High level overview of the GPU threading and scheduling model.

Parallel programming on a GPU uses a "*Single-Program, Multi-thread*" (SPMT) model, as shown in Table 2.2, similar to a *Single-program multi-data* (SPMD) model. A single program, called a *kernel*, is loaded onto each SM (SMX) and then the kernel is executed in parallel across all threads scheduled onto that SM. At every instruction issue, the SM selects an active *warp,* a bundle of *threads,* which is ready to execute and issues the next instruction to the active threads of that warp. Each warp can be thought of as a *fat* thread, where all 32 threads in each warp execute in lock step as they march through the instructions of a kernel program. Each thread within each warp executes the exact same instructions but on different data values stored in thread local registers. This SIMD model approach strongly favors data-level parallelism (DLP). Each unique warp processes its own independent instruction stream, so the SPMT model supports the multi-threaded programming model but at the warp level instead of the thread level. This approach is called the SIMT (*Single Instruction, Multi-Threaded*) model and is conceptually halfway between SIMD and full SMT (*Simultaneous Multi-Threading*) programming. As we will see in chapter 3, warp-level instruction processing impacts performance for conditional branches and loops.

CTA: Since a GPU is a two-level processing device, two levels of multi-threading are needed to fully expose the underlying hardware functionality. On the GPU, threads are organized into a *cooperative thread array* (CTA), a 2 level thread hierarchy that maps onto the physical 2 level processor hierarchy of SMs (including SMXs) and SPs. Typically several GPU kernels are coordinated by a host CPU function to accomplish a single task or algorithm. In each kernel, threads are organized into a fixed CTA. Each CTA *grid* is a 1D or 2D layout of thread *blocks*. Each thread *block* is a 1D, 2D, or 3D layout of up to 1,024 threads. All thread *blocks* in a *grid* must have the same layout ‹size, shape› and number of threads. A high level thread manager, called the *GigaThreads Engine*, at the GPU device level supports coarse-grained scheduling of individual *blocks* in the *grid* onto the physical SM's to keep them all busy. Each SM (SMX) contains a *warp scheduler*, which supports fine-grained parallelism across the SPs on each SM by scheduling active *warps* within each concurrent *block* for execution on their own instruction stream. The warp scheduler supports light weight thread creation, zero-overhead thread scheduling, and intra-block barrier synchronization.

Warps: As noted before, threads are grouped (or divided) into virtual groups of 32 threads called *warps*. (If there are less than 32 threads in a *block* than multiple *blocks* are combined into a single *warp*. If there are more than 32 threads in a *block* than the *block* is sub-divided into multiple *warps*.) Each *warp* within a thread *block* is executed independently from all other warps, each warp can be thought of as fat thread in a traditional multi-threaded programming model. In other words, multi-threading occurs at the *warp* level not at the individual SP thread level in the SIMT model. Each warp in a thread block is analogous to a vector-parallel architecture with a vector length of 32.

Scheduling Constraints: The *GigaThread*s scheduler handles thread blocks within each grid. The scheduler only supports grids with a 1D or 2D thread block layouts with a maximum of 65,535 blocks per dimensions. A warp scheduler on each SM (SMX) handles scheduling warps from a pool of up to 8 (16) concurrent thread blocks. The warp pool hides execution latency by frequent switching of warps as necessary to mitigate long I/O requests and shorter stalls caused by data dependencies.

On the GTX 580, Each SM can schedule up to 8 thread blocks containing an aggregate total of up to 48 warps (1,536 threads) concurrently, provided enough resources exist to support the necessary assigned registers on a per-thread basis and shared memory on a per-block basis. On the GTX 680, Each SMX can schedule up to 16 thread blocks containing an aggregate total of up to 64 warps (2,048 threads) concurrently.

This means that the modern GPU cards can support thousands of threads running in a warp-threaded manner – up to 24,576 and 16,384 threads on the GTX 580 and 680 respectively. GPUs support TLP on a truly massive scale.

Latency Hiding: As mentioned in chapter 2.1.1, Multi-threading is a well-known TLP paradigm for increasing system throughput. GPUs rely heavily on *warp-threading* to hide latency caused by I/O, serialization, or dependencies. Long term I/O latency is caused by memory accesses. The hardware can cause short-term latency via serialization when managing concurrent access. Short term compute latency can also be caused by pipeline stalls due to instruction dependencies. With enough concurrent warps, latencies can be hidden by switching from stalled warps to active warps that keeps the SPs busy doing useful work.

The SM (SMX) cores include fine-grained instruction, control and scheduling logic. The SM (SMX) cores also include large resource pools of registers (32K or 64K), shared memory (48KB), and warp context tables (up to 48 or 64) to support warp-threading of concurrent thread blocks (up to 8 or 16) on each SM. Unlike CPU's, Multi-threading and scheduling on a GPU occur at the warp level not at the level of individual threads. Just like CPU multi-threading, the SM warp scheduler rapidly switches between warp contexts as the running warp context stalls on short pipe-line hazards or long I/O operations. Rapid context switches between stalled and active warps can occur as frequently as once each cycle. This allows each SM (SMX) to hide latency due to stalls by massive fine-grained scheduling across all active warps.

Warp Registers: Each individual thread within a warp bundle is a true thread in the sense that it is executed on its own unique SP core and each thread contains its own unique state (registers) that allows it to work on its own individual data. However they are not as general as threads (POSIX) as we are used to normally thinking of them on regular CPUs. This is because all thread creation, execution, and scheduling occurs in lock step at the warp level not the thread level. Special registers including a *program counter* (PC) are stored on a per warp basis. Some of these special registers keep track of the CTA layout (size and shape) and the starting *block id* (bid) and *thread id* (tid) for use in computing access indices into data arrays. Other special hardware register masks are used to track active/inactive individual threads within each warp to support concepts such as serialization, partially full warps, conditional branching (different threads taking different branch paths) and individual threads taking differing amounts of processing time to finish execution for certain algorithms.

Threading and Memory: Threads may access data from multiple memory spaces during their execution including registers, shared memory (local scratch-pad), constant memory, texture memory, surface memory, local memory, and global memory. Registers are readable and writable but not indexable. Shared memory typically takes 18-22 machine cycles to access and is readable and writeable and indexable. Global memory corresponds to the DDR3 (or DDR5) DRAM on the GPU device itself and typically takes 400-800 cycles to access and is readable and writeable and indexable. A small read-only L1 cache exists on each SM (SMX) and is used to speed-up transfers between global memory and registers. Another larger read-write L2 cache exists between global and shared memory. Constant and texture memory are read-only. Surface-memory is readable and writable. Local memory as mentioned before is a platform behavior to handle register spills.

High performance API calls are available to transfer data arrays between CPU RAM and GPU RAM using *direct memory access* (DMA). Current maximum transfer speeds range from 4 – 8 GB/s based on the underlying host CPU/Memory architecture. These transfers can be done synchronously or asynchronously.

Although GPUs can support the main parallel programming models (task parallelism, pipelining, and data parallelism) they are especially well-suited to problems that can be expressed as data-parallel computations. Data-level parallelism partitions the data domain and then maps those partitions onto parallel processing threads. On GPUs, these processing threads are in turn are mapped onto SPs. Each parallel thread plus its associated state (registers) represent the smallest unit of execution on the GPU.

2.2.3 Parallel Coordination

GPU architects have provided only limited support for coordination between threads. Coordination is the behavioral cooperation across threads of a parallel algorithm needed to achieve correct results. Coordination includes the concept of communicating intermediate and final data results between threads. Typically on GPUs, coordination is only supported between threads belonging to the same thread block. The coordination mechanisms between threads are as follows -- memory, atomics, voting intrinsics, and barrier synchronization.

Memory: Although GPUs support multiple types of memory including registers, shared memory, and global memory. As we will see shortly, shared memory is the best fit for coordination/communication between threads. *Registers*: Each thread is only allowed to access registers that have been assigned to it by the CUDA compiler and GPU hardware. In addition, registers are not addressable, in other words, thread registers cannot be accessed using index operations. As a result, registers cannot be used to coordinate threads. Note: The new CC 3.5 Kepler architecture has support for a new PTX "shuffle" command that allows the threads within a single warp to move registers values between threads. *Shared Memory*: Unlike registers, shared memory is addressable and accessible by all threads within the same thread block. Shared memory can be used to coordinate threads by storing common results or behavioral state. However, coordination/communication across thread blocks within a grid cannot be done using shared memory. Since shared memory is visible to all warps belonging to the same thread block and concurrently running warps can compete to access the same memory resources, programmers need to ensure mutual exclusion between different warps for correct parallel behavior. *Global Memory*: Although global memory can be used to store common results or state for all threads within a thread block, it runs much slower than shared memory. Similar to shared memory, programmers need to ensure mutual exclusion between different warps and blocks when accessing global memory used for communication/coordination. It is difficult to coordinate behavior across thread blocks within a grid using global memory due to two main factors 1) the non-deterministic thread block schedule generated by the giga-engine scheduler 2) New thread blocks are not scheduled onto SMs until currently running thread blocks have completed. As a result, using global memory to coordinate thread blocks across a grid is not recommended.

Atomics: An *atomic operation* is a small group of hardware instructions guaranteed to appear as if the entire group was executed as a single indivisible instruction by the rest of the system. Once an atomic operation has started other threads cannot interrupt the current thread until the atomic operation has successfully completed. As a result, atomic operations can be used to ensure mutual-exclusion when accessing common resources in shared or global memory by multiple threads (warps or blocks). However, the GPU hardware serializes thread access, which negatively impacts parallel performance. Consider, tens of thousands of threads competing to update the exact same memory address using atomics. Instead of tens of thousands of threads executing concurrently, they all now must execute sequentially. As a result, using atomics must be done carefully to avoid degrading performance due to the massive parallel overhead caused by hardware serialization. **Note:** Atomic operations on the Kepler architecture execute faster than on the Fermi architecture.

Voting Intrinsics: The GPU implements serialization by predication hardware that supports gathering Boolean predicates across all 32 threads within a warp into a single 32-bit mask. That predicate mask is then shared across all threads within a warp. NVIDIA has exposed this hardware functionality to programmers as voting intrinsics in the ISA. This allows the threads within each warp to communicate the results of simple predicate {*true, false*} tests with each other across registers in a single instruction without first needing to store that information in slower shared memory.

Barriers: The CUDA platform supports single-instruction *barrier* commands that force all *warps* within a thread *block* to be synchronized (coordinated) at a single check-point before any warp starts back up doing useful work. This synchronization helps support correct behavior of algorithms using multiple warps per-thread block. The parallel threads within a single *warp* do not need any barrier synchronization for correct behavior since they already move in lock-step according to their SIMD vector-parallel design.

Each GPU card supports moderate ILP, large DLP, massive TLP, and a complex memory hierarchy. ILP is supported via a 2-level MIMD/SIMD processing cores. The SM warp cores support pipelining, scoreboarding (Fermi Only), and multi-issue. DLP is supported via SP core replication within each SM core and then SM core replication within the GPU card. TLP is supported by 2-level warp-threading. The memory hierarchy in decreasing access speed is registers, shared memory, global memory, and CPU RAM. Fixed sizes on memory and execution contexts put constraints on how programmers exploit data-level parallelism in their algorithms. All this complexity plus constraints results in many issues that make it hard for GPU programmers to write correct, robust, and fast code. In the next chapter, I will present some of the main issues that GPU programmers should be concerned about.

3.0 Performance and Issues Hindering Performance

In my case studies, I will show how the contrast between the CPUs latency focus and the GPUs throughput focus (Garland and Kirk, 2010) leads to different choices for programmers as they maps tasks onto these different architectures. CPUs were originally designed to solve a general set of computing tasks, often with direct user interaction. Consequently, CPU design has concentrated on reducing *latency*, which is the time duration between a user request and computer response. In contrast, GPUs were originally designed to take a large stream of geometric data elements, perform essentially the same computation on each element (for example, rotate, illuminate, clip, project, or rasterize it), and produce 30-60 rendered frames per second. In other words, GPU design has focused on maximizing *throughput*, which is the ratio of the number of work items processed over some period of time (for example, triangles rendered per second).

Programmers need to be aware of this throughput design focus on GPUs, in order to achieve high performance with GPU programming. So, in this chapter, I first describe various performance metrics that I use for measuring throughput. After that, I categorize the main issues affecting throughput that crop up in my case studies into three broad groups: Parallel Performance Issues, GPU Architecture Issues, and GPU Memory Issues.

3.1 Measuring Performance Throughput

To understand throughput and performance bottlenecks, we first need to be able to measure performance accurately. The GPU platform provides machine counters, from which metrics for speedup and throughput can be derived. In my case studies, I concentrate on three throughput metrics: instruction, I/O, and data throughput. Other performance metrics (such as total cycles, speedup, and work and depth analysis) will also be used in my case studies but will not be the focus.

To understand what are the bottleneck issues affecting performance, programmers start with certain known *values*, then take experimental *measurements* to derive *metrics* that provide insight. Values known by programmers include the input size (*n*) and output size (*m*). The CUDA profiler or hardware timers record useful measurements such as timings (*time*), number of parallel cores (*p*), and total threads (*t*). NVidia GPUs also provide various machine counters for profiling performance, including *instructions issued* (II) and the average *instructions* retired *per* machine *cycle* (IPC). From these basic values and measurements, I compute derived metrics such as *total cycles*, *TC* = *II*/*IPC*, which measures the total number of machine cycles to complete a section of code, algorithm, or an entire program.

3.1.1. Throughput Metrics

In my case studies, I consistently use three throughput metrics to gauge algorithmic performance. *Instruction throughput* (MI/s or GI/s, meaning mega- or giga- instructions executed per second, over all threads) tracks algorithmic performance. *I/O throughput* (MB/s or GB/s, meaning mega-bytes or giga-bytes transferred per second) tracks memory transfer performance. *Data throughput* (M*/s or G*/s, meaning mega-units or giga-units handled per second) tracks algorithmic performance in data units most germane to the problem space. Note that each of the three is a simple ratio of basic measurements.

A typical throughput graph, as seen in Figure 3.1, plots throughput on the y-axis as a function of input size (n) on the x-axis (usually in log-scale).

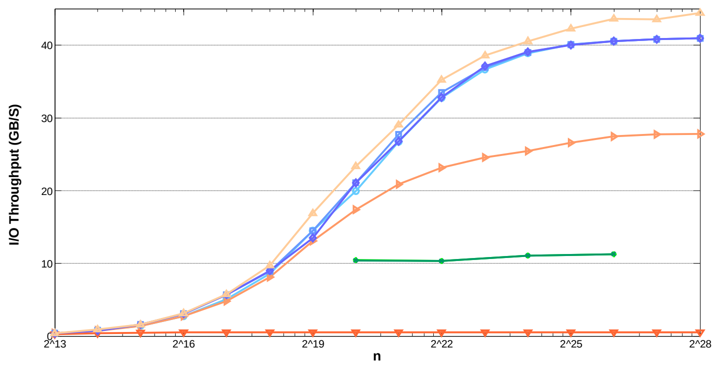


Figure 3.1: Histogram I/O-throughput

Like Figure 3.1, most GPU throughput graphs have sigmoidal ("S" shaped) curves for increasing values of input sizes (n). The throughput performance curve starts off flat for small input sizes ($n \leq 10^3$), grows rapidly for medium input sizes ($10^3 < n \leq 10^6$), and then levels off at some fraction of the hardware's peak throughput for large input sizes, ($10^6 < n$). For small input sizes, there is not enough data to use the parallel hardware efficiently or to amortize the heavy GPU kernel launch costs. For large data sizes, there is enough data to parallelize work across tens of thousands of threads and the initial kernel launch costs are amortized across millions of data elements decreasing launch costs to a negligible fraction of total performance. For medium data sizes, throughput performance transitions from the inefficient to efficient case as the input size increases.

3.1.2 Parallel Speedup and *Work* and *Depth* Analysis

In addition to throughput metrics, there are two other traditional notions of performance for parallel computation that I use in my case studies: parallel speedup and work and depth analysis (Hennessey and Patterson, 2010). Let me introduce them by analogy to their serial equivalents: serial speedup and asymptotic runtime analysis.

The concept of *speedup*, $S = \left(\frac{Time_{old}}{Time_{new}}\right)$, allows us to compare the performance of two similar programs, algorithms, or sections of code using simple timings, with one timing for the old code and one timing for the new code. *Serial speedup* (SS) is the ratio of the amount of time it takes to complete a task using an improved program versus a baseline program, $SS = \frac{Time_{baseline}}{Time_{improved}}$. *Parallel speedup* (PS) is the ratio of the time to complete a task using a serial computation versus the time to complete the same task using a parallel computation, $PS = \frac{Time_{serial}}{Time_{parallel}}$.

Often a serial algorithm cannot be fully parallelized due to unavoidable dependencies between sections of code. *Amdahl's Law* (Amdahl, 1967) predicts the theoretical maximum parallel speedup on p processors for a specific problem for which a fraction of the program $\alpha \in [0,1]$ is inherently sequential and the rest of the program $(1-\alpha)$ is parallelizable: $S(\alpha, p) = \left(\frac{1}{\alpha + \frac{1-\alpha}{p}}\right)$. Even given an infinite number of parallel processors, total performance cannot exceed $\left(\frac{1}{\propto}\right)$. In other words, total performance is constrained by the serial portion of the program. Amdahl's law, which can be derived from parallel speedup by normalizing serial time to one, suggests that programmers focused on latency can solve a fixed-sized problem in the shortest period of time by removing serial constraints.

As a counterpoint, Gustafson observes that end-users often exploit the maximum computing power available to them to solve ever larger problems over some practical time period (minutes, hours, days). *Gustafson's Law* (Gustafson, 1988) as $S(\alpha, p) = \alpha + (1-\alpha)p$ suggests that programmers focused on throughput issues can push more work through the system using massive parallelism. Gustafson's law can be derived from scaled parallel speedup by normalizing parallel time to one.

Asymptotic analysis, also known as "Big '*O*' notation", shows how the running time (or resource usage) of an algorithm grows with the input size (*n*). The growth-rate function is reported using asymptotic notation to suppress implementation-dependent constants and to simplify expressions. If these hidden constants are reasonable, then $\boldsymbol{O}(\log n) \ll \boldsymbol{O}(n) \ll$

$\boldsymbol{O}(n \log n) \ll \boldsymbol{O}(n^2)$. In words, a logarithmic growth-rate algorithm is preferred to a linear growth-rate, a linear growth-rate is preferred to a log-linear growth-rate, and log-linear growth-rate is preferred to a quadratic growth-rate algorithm.

Work + Depth Analysis: For a parallel computation, *work efficiency* $W(n)$, is defined as the total number of instructions executed across all parallel cores. The ratio of work to the number of cores (p) results in the linear *ideal speedup* $\left(\frac{W(n)}{p}\right)$ of a parallel computation. However, there are usually dependent steps, either inherent in the algorithm itself or with the coordination required between parallel cores, which limit this ideal speedup. The longest dependent chain of executed steps on any core is defined as an algorithm's *depth efficiency* $D(n)$. Figure 3.2 illustrates work and depth for summation, in both serial and parallel forms.

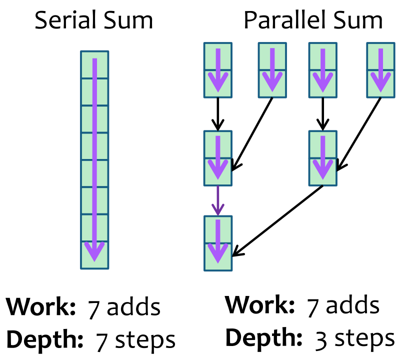


Figure 3.2: An example of work and depth efficiency for the simple summation of 8 values for both the serial and parallel cases.

Work efficiency and depth efficiency are related by *Brent's Theorem* (Gustafson, 2011), which says that any parallel algorithm runs in $\frac{W(n)}{p} + D(n)$ time, in fact: $max\left(\frac{W(n)}{p}, D(n)\right) \leq Time \leq \frac{W(n)}{p} + D(n)$. Four useful implications from Brent's Theorem are

- A parallel algorithm cannot run faster than its depth efficiency, $D(n)$.
- A parallel algorithm that requires coordination across p cores cannot run faster than $O(\log p)$.
- It is inefficient to use more parallel cores than you need to solve an algorithm. The concept of *Parallelism* $\left(\frac{W(n)}{D(n)}\right)$ roughly captures how many parallel cores an algorithm can efficiently use.

- If the number of processors and input size are fixed or known ahead of time than try to do at least $D(n)$ work on each processor for the best parallel efficiency.

3.2 Parallel Performance Issues

Metrics are useful in measuring performance and understanding the issues that impact performance. There are three main issues that impact parallel performance: scalability, parallel overhead, and load balancing.

3.2.1 Scalability

A system is scalable if adding parallel resources (such as computers, GPUs, cores, and threads) increases parallel performance. Users prefer systems that are scalable because they can simply add more hardware resources over time to deal with increasing computational demands. In my case studies, we will see that I prefer data-level parallelism almost exclusively over task-level parallelism because the former scales readily for increasing data sizes.

3.2.2 Parallel Overhead

Programmers should minimize coordination costs across parallel threads to achieve better performance. Parallel overhead is the ratio of the time spent coordinating parallel work between threads versus the total time it takes to solve the original problem. A system has low parallel overhead if the amount of time spent coordinating algorithmic behavior across parallel threads is small. A system has high parallel overhead if the amount of time spent coordinating across threads is large. For better parallel throughput, programmers should minimize parallel overhead. As mentioned in Section 2.2.3, “Parallel Coordination,” for GPU architectures, there is only limited hardware support for coordination mechanisms between threads.

3.2.3 Load Balancing

Programmers often seek to take full advantage of the hundreds of processing cores available on a GPU card by keeping them all busy. Parallel utilization is a measure of how many parallel cores out of all cores in a system are busy doing useful work at any given time. Keeping cores busy can be accomplished by load-balancing work evenly across hundreds or thousands of threads. A system has poor load-balancing if many cores are kept idle for long periods of time or if many cores finish early while some cores still have considerable work to do. A system has good load balancing if most of the cores are kept busy most of the time and all the cores finish executing at approximately the same time.

Although there are many techniques to achieve optimal load-balancing,[1] in my thesis, I focus exclusively on *equal partitioning*, an approach which divides (n) data elements across (p) threads into approximately equal sized runs $\lceil n/p \rceil$. Equal partitioning works well because it supports SIMD processing on GPUs, recall that all SPs on each SM move in lock-step through a single instruction stream. Since GPUs use a more complicated 2-level thread hierarchy for scheduling (as we will see in my case studies), I adapt equal partitioning as equal-sized runs on fixed-size data blocks.

Since, GPU programmers often assign one thread per work item for simplicity. For better performance, it is often useful to go even further and use multiple work-items per-thread (*over-decomposition*) and multiple thread warps per core (*over-subscription*) to help hide stalls on each core that would otherwise decrease parallel throughput. In my case studies, you will see that load balancing is readily achieved via data-level parallelism by using a large enough input array to saturate the processing resources.

3.3 GPU Architecture Issues

Five GPU architecture issues also impact parallel performance: CTA partitioning, GPU scheduling—stalls and hazards, GPU scheduling—constraints and occupancy, multi-issue dispatch, and branch divergence.

[1] Such as dynamic work assignment, work stealing, or equal partitioning

3.3.1 CTA Partitioning

Recall that a *cooperative thread array* (CTA) is a 2-level hierarchy (thread blocks within a grid and threads within a thread block.) In order to take advantage of the massive data-level parallelism via threads on a GPU, GPU programmers need a way to specify the desired thread structure and thread to data mapping. This is done with a 2-step process that I call *CTA partitioning.*

1) For each GPU kernel launch, GPU programmers must structure the desired shape and size of the CTA layout using two 3D parameters (grid and block) both specified as 3-tuples ‹*x*,*y*,*z*›.
2) Inside each GPU kernel, GPU programmers must map CTA layout parameters that represent unique threads onto actual data locations. The CTA layout parameters are available to programmers inside each GPU kernel via four global read-only 3D parameters (`gridDim`, `blockDim`, `blockIdx`, `threadIdx`) as 3-tuples ‹*x*,*y*,*z*›.

When I was first introduced to GPU programming, I found the CTA layout and mapping steps confusing, so, I present more details on how to perform the CTA layout and mapping steps below.

CTA layout: CUDA requires that the CTA layout must be specified as part of each GPU kernel launch. The grid and block CTA layout parameters are specified as 3-tuples ‹*x*,*y*,*z*› that represent the ‹width, height, length› of a 3D grid or thread block layout. To refer to both the *g*rid and block dimensions as one unit, I combine both as a 6-tuple denoted as ‹*gw*,*gh*,*gl*, *bw*,*bh*,*bl*›. Various constraints on the maximum size and shape of these grid and block CTA layout parameters must be taken into consideration. Each tuple value is specified as a 32-bit signed integer. Zero and negative values do not make physical sense. Therefore, the valid range for any tuple value is $[1,2^{31}\text{-}1)$. Programmers can eliminate unwanted dimensions by specifying a default value of one (1) for the unused value. This specification enables 1D, 2D, and 3D layouts, as desired. CUDA currently does not use the *gl* (*grid.z*) CTA dimension value in any way. Consequently, the actual CTA layout is effectively ‹*gw*,*gh*,1, *bw*,*bh*,*bl*›. The Fermi architecture only supports a maximum value of 65,535 (2^{16}-1) for any dimension for the grid parameter. As a result, the valid range is limited to [1,2^{16}-1), whereas the Kepler architecture supports the full range [1,2^{31}-1). For Fermi, this limitation means that if more than 65,535 thread blocks within a grid are needed, then a 2D grid layout is the only solution to fully cover all the desired thread blocks. CUDA currently limits the thread block size (TBS) to 1,024 threads per-block or less ($bw \cdot bh \cdot bl \leq 1{,}024$). Since thread processing is actually done in warp-sized batches, the TBS should be a multiple of the warp size (meaning, 0 == *TBS* % 32). Processing thread blocks in warp sized batches helps keep threads busy (as well as the SP cores that execute them) within each thread block. The `grid` CTA layout parameter supports 1D and 2D layouts (thread blocks within a grid), while the `block` CTA layout parameter supports 1D, 2D, and 3D layouts (individual threads within a thread block). Once specified (at launch time), the grid and thread block layouts remain fixed for the entire execution of a specific GPU kernel.

CTA mapping: Inside of each GPU kernel, programmers must map individual threads onto data locations. To help with this mapping process, CUDA provides four global read-only CTA parameters as 3-tuples (`gridDim`, `blockIdx`, `blockDim`, `threadIdx`). These CUDA variables are always available for use by the code anywhere inside the CUDA kernel, including nested function calls. The `gridDim` and `blockDim` parameters refer back to the original CTA layout (size and shape) parameters specified at kernel launch, with the ‹*x*,*y*,*z*› dimension values meaing ‹width, height, length›, respectively. For convenience, I represent both the `gridDim` and `blockDim` variables as a single unit, ‹*gw*,*gh*,1, *bw*,*bh*,*bl*›. The `blockIdx` and `threadIdx` parameters uniquely identify the currently running thread block (within the current grid) and the currently running thread (within the current thread block). The `blockIdx` and `threadIdx` can be thought of as a multi-dimensional block ID (*bid*) and thread ID (*tid*), respectively. These variables uniquely identify the location of each individual thread within the structured thread hierarchy. Again, for convenience, I represent both the `blockIdx` and `threadIdx` parameters as a single 6-tuple as ‹*bx*,*by*,*bz*, *tx*,*ty*,*tx*›. All four CTA parameters help GPU programmers map individual threads onto their corresponding data items.

As an example, Figure 3.3 contains a code snippet that shows a full 5-dimensional CTA mapping down onto a single unique thread index.

```
// Map Block ID (bid) from <gridDims, blockIdx>
 gW = gridDims.x     gH = gridDims.y    // gL = gridDims.z (not used)
 bX = blockIdx.x     bY = blockIdx.y    // bZ = blockIdx.z (not used)
bid = (gW*bY) + bX     // 2D to 1D mapping
// Map Thread ID (tid) from <blockDims, threadIdx>
 bW = blockDims.y   bH = blockDims.y   bL = blockDims.z
 tX = threadIdx.x   tY = threadIdx.y   tZ = threadIdx.z
tid = (bH*bW*tZ) + (bW*tY) + tX  // 3D to 1D mapping
// Map Thread Index from <bid, tid>
 TBS = bH*bW*bL        // Thread Block Size
tIdx = (bid*TBS)+tid   // 2D to 1D mapping
// Map Data Offset from <tIdx>
dataOff = ...          // Problem Domain Specific
```

Figure 3.3: A simple mapping from the four CTA layout parameters onto unique block and thread IDs that in turn are mapped onto a unique thread index. Note: This is only one of many possible ways to map CTA layout parameters onto a unique thread index within the CTA.

The code snippet, as shown in Figure 3.3, has four main steps: First, it maps the 2D grid layout onto a unique block ID (*bid*) within the grid. Second, it maps the 3D block layout onto a unique thread ID (*tid*) within the thread block. Third, it maps the *bid* and *tid* onto a unique thread index (*tIdx*) within the entire CTA. Finally, it maps the thread index onto the specific data offset that needs to be processed by this thread. This step is not shown as this must be defined by the GPU programmer for their specific problem. CTA layout configurations that use fewer dimensions require less total operations in order to do the mapping.

Flexibility in specifying the CTA partitioning (layout and map) allows programmers to choose data layouts that best fit their problem domain: 1D, 2D, or 3D. This freedom of choice becomes a burden; however. Before programmers can begin to code, they are forced to make choices on how to structure the CTA layout, how to partition data across the threads, and how to map threads onto data inside of each kernel. These choices have big effects on performance, but whether these effects will be positive or negative is initially unclear. In my case studies, my data access skeletons help me explore choices for CTA partitioning and their effects (see Chapter 5 “Data Access Skeletons” for more details).

3.3.2 GPU Scheduler — Stalls and Hazards

Since GPUs are throughput oriented programming devices, our goal is to keep the hundreds of processors on each GPU device as busy as possible. Processor stalls, where the processors sit idle instead of doing useful work, are to be avoided. GPU programmers should realize that in a GPU almost every instruction fetch, register, or memory access is slower than a processor cycle. The latency of these operations causes an *instruction stall* while each instruction waits on its input. During this instruction stall, the compiler and scheduler together try to ensure that a thread warp has other useful work to do in order to avoid a *thread stall*. Thread stalls are frequent; so the GPU scheduler has many tricks to quickly swap in another thread warp to execute in order to avoid a more grievous *processor stall*.

GPU programmers aiming for high throughput need to write code that the compiler and scheduler can exploit to avoid processor stalls. GPU Programmers also need to provide massive numbers of threads so that the processors can switch to other thread warps to avoid processor stalls. So, let’s focus on preventing thread stalls. It helps to know that there are two main types of thread stalls – short term and long term.

Short terms stalls of just a few cycles typically come from compiler- or scheduler-induced delays[2] inserted to avoid structural, control, or data hazards that otherwise could jeopardize correct program behavior. *Structural hazards* occur when two or more stages in a pipelined or out-of-order instruction architecture compete for the same functional units, such as

[2] These delays are often handled by inserting several wasteful *no operation* (NOP) commands into the instruction stream before inserting the delayed instruction that does useful work.

adders or loaders. These hazards are typically resolved by chip architects using replication, such as putting a full adder in each competing pipelined execution stage. *Control hazards* occur when the code executes branch or loop instructions. In these situations, the scheduler needs to choose between two paths {*true, false*} but does not know which path to take until the branch outcome is known. Picking a branch outcome leads to a branch delay, which is the number of machine cycles it takes from starting the original branch instruction until the actual branch outcome *true* or *false* is known. *Data hazards* occur when one instruction accesses a register that another concurrently running instruction also needs to access. Ignoring data hazards can lead to incorrect program results.

There are three main types of data hazards: read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW)[3]. WAR and WAW dependencies, where two instructions happen to use the same register (aliasing), can be resolved by assigning different registers to the competing variable names in order to remove the dependency[4]. A RAW dependency means that one instruction consumes the output of a prior instruction as input and thus must wait until the prior instruction completes. The RAW dependency is real and must be honored for correct program behavior.

Short term stalls can be avoided by the compiler and scheduler if they can find other independent instructions in the same warp's instruction stream that can be safely executed as the current instruction stalls. Programmers can help by writing code with good Instruction-level Parallelism (ILP).

Long term stalls of hundreds of cycles typically come from servicing I/O transfer requests. When this kind of stall occurs, the hardware scheduler on each SM can hide latency by rapidly switching, up to once per cycle, a stalled thread warp for another thread warp that is ready

[3] These RAW, WAR, WAW data hazards are also known as true-, anti-, and output-dependencies respectively.
[4] *Register renaming* is the original term used by Tomasulo (Hennessey and Patterson, 2012) to describe how to remove WAR and WAW data hazards caused by aliasing different variables on the same register.

to execute. Programmers can help by writing code that launches thousands of concurrent threads to support Thread-level Parallelism (TLP). To better understand how switching can hide latency via TLP techniques, assume one load instruction issued per warp and that loading a value from shared memory has a latency of twenty machine cycles (a reasonable rule of thumb on Kepler). Under these conditions, a GPU programmer would need to schedule at least twenty warps per core to have enough in-flight load commands to keep each core fully busy. NVIDIA measures this with occupancy, which is discussed later on in this chapter. Alternatively, programmers could hide the same latency using ILP, by scheduling at least three thread warps per core and then having each warp issue eight independent load commands. This approach would result in 24 in-flight loads, which would completely engage each SM core. So, scheduling more thread warps hides stall latency via TLP, and doing more work per thread hides stall latency via ILP.

NVidia defines the term *occupancy*, which I think of as a rough measure of the potential for using TLP to hide latency. In general, kernels with higher occupancy tend to have better performance because, when a current warp stalls, they give the SM warp scheduler more active warps to choose from. Because ILP can also hide latency, as just illustrated in my switching example, Volkov (Volkov, 2010) advises that performance of some kernels can actually decrease as occupancy increases above 50%. My case studies will show that experiments are needed to balance contributions from ILP and TLP and achieve best performance.

3.3.3 GPU Scheduler — Constraints on Occupancy

NVIDIA defines *occupancy* for a kernel as the ratio of the number of active thread warps scheduled on an SM to the direct limit of active warps for that SM type. Although a GPU programmer's CTA layout may specify tens of thousands of thread blocks (and thread warps), each on-chip SM scheduler can track and process only a small batch of active thread blocks at any time. The SM scheduler not only has direct limits[5] on the number of active blocks and warps, but it also has indirect limits on them due to constraints on register and shared memory pool sizes.

Register Pool Size Constraints: Each SM has a limited pool of 32-bit registers to distribute across all active threads[6]. This register pool size is 32 K on Fermi architectures and 64 K on Kepler. These pools may seem large until each pool is divided across hundreds of active threads. The SM warp scheduler cannot exceed the register pool size and will throttle back the number of thread blocks that it makes *active* (i.e. allows to be concurrent at any given time on each SM). For instance, suppose that the GPU programmer chooses to have 128 threads per block, and the CUDA compiler decides that the code needs 37 registers per active thread. On the Fermi architecture, the scheduler can make at most six thread blocks active since $(32K)/(128 \cdot 37) = 6.91$, which limits Fermi occupancy to 50%, since $(4 \text{ warps} \times 6 \text{ blocks})/48 \text{ max.} = 24/48$. On the Kepler architecture, the scheduler can make no more than 13 thread blocks active, since $(64K)/(128 \cdot 37) = 13.84$, which limits Kepler occupancy to 81.25%.

Shared Memory Pool Size Constraints: Each SM has a limited pool of shared memory spread across all active thread blocks. This pool has a maximum limit of 48 KB on both Fermi and Kepler architectures. Again, 48 KB seems large until you divide it across all active thread blocks. (In fact, the programmer can also choose to trade the shared memory pool size in 16KB increments for L1 cache size.) The scheduler cannot exceed the shared memory pool size and will decrease the number of thread blocks that it makes active. Suppose for example, that a GPU programmer writes a kernel that consumes 10 KB of shared memory for a thread block of 128 threads. On both the Fermi and Kepler architectures, the scheduler can make active no more than four active thread blocks, since 4 = (48 KB per SM / 10 KB per block = 4.8), limiting occupancy to 33% (16/48) on Fermi and 25% (16/64) on Kepler respectively.

[5] On Fermi, *MaxBlocks* = 8 and *MaxWarps* = 48; on Kepler, 16 and 64. CTA parameters determine which of the direct limits, *MaxWarps* or *MaxBlocks*, serves as the hard limit on the SM warp scheduler, as we will see in case studies.
[6] Fermi and Kepler architectures further limit the maximum number of registers allowed per thread to 63; The GTX Titan architecture extends this to 255 (NVIDIA, 2012, CUDA Programming Guide).

Constraints on Occupancy (Fermi)						**Bytes**	**Bytes**
Chosen Threads/block	***Max.Occu-pancy %***	***Max. Blocks***	***Max. Warps***	***Max. Threads***	***Max regs /thread***	***Max Sh.Mem. /block***	***Max Sh.Mem. /thread***
32	.17	8*	8	256	63*	6,144	192
64	.33	8*	16	512	63*	6,144	96
128	.67	8*	32	1,024	32	6,144	48
256	1.00	6	48	1,536	21	8,192	32
512	1.00	3	48	1,536	21	16,384	32
1,024	.67	1*	32	1,024	32	49,152	48
Constraints on Occupancy (Kepler)						**Bytes**	**Bytes**
Chosen Threads/block	***Max.Occu-pancy %***	***Max. Blocks***	***Max. Warps***	***Max. Threads***	***Max regs /thread***	***Max Sh.Mem. /block***	***Max Sh.Mem. /thread***
32	.25	16*	16	512	63*	3,072	96
64	.50	16*	32	1,024	63*	3,072	48
128	1.00	16	64	2,048	32	3,072	24
256	1.00	8	64	2,048	32	6,144	24
512	1.00	4	64	2,048	32	12,288	24
1,024	1.00	2	64	2,048	32	24,576	24

Table 3.1: Constraints on Occupancy (for both Fermi and Kepler architectures).

Table 3.1 shows the maximum occupancy possible given a chosen number of threads per thread block for Fermi and Kepler architectures. It also shows the maximum registers per-thread and maximum shared memory usage per thread block to achieve maximum occupancy. Data layout and kernel implementation often make it difficult to stay under these resource limits.

As we will see in the Reduce/Scan case study in Chapter 6, GPU programmers must consider constraints on occupancy when choosing the initial CTA thread block and grid sizes in order to balance active thread blocks evenly across the available SMs on each GPU card.

3.3.4 Multi-issue Dispatch

Recall that both the Fermi and Kepler architectures support multi-issue dispatch[7], meaning the idea that one SM core can dispatch up to *k* simultaneous instructions per clock cycle in parallel from *k* separate thread warps. The *k* independent instructions are dispatched and executed in parallel on multiple redundant function processing units (many ALUs and FPUs per

[7] Also known as *simultaneous multi-threading* (SMT). This is a form of thread level parallelism.

SM core). Multi-issue dispatch is much like having multiple assembly lines running on one factory floor at the same time.

On Fermi architectures, such as the GTX 580, each SM core supports dual issue. This means that each SM can schedule and execute up to two thread warps concurrently as long as there are no dependencies between the two warps. On Kepler architectures, such the GTX Titan, each SMX core supports quad issue, meaning each SMX can schedule and execute up to four thread warps concurrently. In addition, Each Kepler SMX warp scheduler is designed to issue up to two independent instructions per cycle from each warps instruction stream[8].

Because of the way these architectures schedule and issue warps, programmers, for better efficiency, should aim to schedule at least two warps on Fermi per SM or four concurrent warps on Kepler per SMX. Since Kepler architectures can also issue up to two independent instructions per cycle, there is a strong incentive for programmers to write code that mitigates data dependencies between instructions so that the SMX warp scheduler can fully exploit the multi-issue feature on each core.

3.3.5 Branch Divergence

GPU programmers should seek to reduce the number of branches and loops used in GPU kernels. Because all threads within a warp step through a single instruction stream in lockstep, *branch divergence* can occur when some threads within a warp take the *true* branch path and the rest of the threads take the *false* branch path. GPU hardware currently solves branch divergence via serialization by predication.

Serialization means that all code from the *true* path and all code from the *false* path are executed sequentially, the code from the *true* path being processed first and the code from the *false* path second. Predication means that each warp maintains a 32-bit predicate mask of Booleans that represents the branch outcomes for the individual threads within the warp. Each

[8] A chip architecture that is able to issue up to *k* parallel instructions from one instruction stream is known as *superscalar*. This is a form of instruction level parallelism.

thread with a predicate mask value of *true* executes instructions only from the *true* branch path while each thread with a predicate mask value of *false* executes instructions only from the *false* branch path.

Serialization by predication can decrease performance by reducing parallelism. From a thread's perspective, a given branch instruction in a CPU needs fetch instructions only for the taken branch path (*true* **or** *false*); whereas, a GPU thread must fetch instructions for both branch paths (*true* **and** *false*). Each nested branch may degrade performance by another factor of two as threads in a warp continue to diverge. In the worst case where branches are nested five levels deep, there could be up to 32 individual sub-branch paths that are executed serially, one for each thread in the warp. This approach results in up to 32× slower performance.

To help avoid some of the performance loss due to branch divergence, each SM scheduler detects when the predicate mask is set to {*all-true*} or {*all-false*}, and it will then behave like a CPU branch instruction, executing only the taken path. Fully efficient warp branching is achieved only when there is no branch divergence within a warp, when all threads within each warp follow the exact same branch path through the code.

On both CPUs and GPUs, avoiding or reducing branches and loops is important to prevent control hazards. On GPUs, avoiding or reducing branches and loops also prevents branch divergence. In my "*k*d-tree" case study (Chapter 7), we will see the negative impact of high branch divergence on processor utilization.

3.4 GPU Memory Issues

Memory constraints, register spills, coalescence, and bank conflicts are all GPU memory architecture issues that impact performance.

3.4.1 Memory Constraints

GPU programmers aiming for high performance must understand several concepts related to memory: access times, limited capacity, and cache constraints. As discussed in Chapter 2, GPU memory is a multi-level hierarchy of registers, shared memory, and global memory. The GPU memory architecture includes hardware support for L1 and L2 caching and as well as for specialized types of memory, such as constant, texture, and local. There is also support to transfer data to and from host memory (CPU RAM) using DMA.

***Access Speeds*:** The consecutive levels of memory have access times that differ by an order of magnitude: Registers ≪ Shared memory ≪ Global Memory ≪ CPU RAM. According to Volkov (Volkov 2010), on the GTX 480 registers are 6× faster than shared memory, which in turn is 7.6× faster than global memory, which in turn is 11.1× faster than CPU RAM[9].

***Limited Capacity*:** Registers are a scarce resource. Fermi architectures support a pool of only 32 K registers; Kepler, a pool of 64 K. Furthermore, both architectures allow a maximum of 63 registers per thread[10]. GPU programmers should expect their programs to have between 21–63 registers available per thread on Fermi and [32-63] registers per thread on Kepler, depending on the number of threads scheduled. Shared memory is also scarce since only 16, 32, or 48 KB is available on each SM (or SMX), and it must be shared across all thread blocks concurrently running on each SM[11]. Global memory

[9] Vasiley Volkov in a talk called “Better Performance at Lower Occupancy” given at the GPU Technology Conference in 2010 (Volkov, 2010) compared the bandwidths of registers (8 TB/s), shared memory (1.3 TB/s), and global memory (177 GB/s) on the GTX 480. This results in speedups of 6× (registers over shared memory) and 7.6× (shared memory over global memory). CPU RAM from the same generation of PC’s had a quoted maximum throughput of 16 GB/s resulting in the claimed speedup of 11.1× (global memory over CPU RAM).

[10] Most Kepler devices (CC 3.0) support a maximum of 63 registers per thread, whereas the GTX Titan (CC 3.5) supports a maximum of 255 registers per thread.

[11] Each SM can have up to eight concurrent thread blocks per SM running at the same time. Each SMX can have up to sixteen.

capacities, on the other hand, vary between 1 and 6 GB, depending on the specific GPU card.

Cache Constraints: Fermi and Kepler GPU architectures both support caching but for a limited number of memory controllers and small caches. The number of memory controllers per GPU varies across different cards, but it is fixed for each specific card and is typically in the range of one to six memory controllers per card. The L1 and L2 cache sizes are also limited, with the read-only L1 cache having [16, 32, or 48 KB] per SM and the read/write L2 cache only having 64-128 KB per memory controller. Since there are up to 16 SMs on Fermi class cards and up to 14 SMXs on Kepler class cards but only 2-6 memory controllers per card, the SMs must compete to use the memory controllers. As a result, data in GPU caches gets evicted much more frequently data in CPU caches. GPU programmers should not depend on their data staying in either GPU cache for long. They may therefore wish to migrate essential data into registers, shared memory, or both.

3.4.2 Register Spills

Although programmers often write code as if they have unlimited registers, real CPUs and GPUs have a limited number of registers available for each thread. On GPUs, the maximum number of registers available is set up at kernel launch time based on the register pool size (32 K on Fermi or 64 K on Kepler) and the number of concurrent threads running on each SM (SMX) core. Compilers dictate the number of registers a kernel needs; register spills occur when the need exceeds what is available. CPU compilers often store spilled variables onto stack or heap memory. The GPU CUDA compiler stores spilled variables into local memory, which is a reserved area of global memory, which gives orders of magnitude slower access. In some case studies, I hand-code key routines just to avoid expensive spills.

3.4.3 Coalescence

When the 32 threads in a warp request to access global memory, the GPU memory controller must transfer the requested data to or from registers in the SM that are assigned to the respective threads. The memory controller does this transfer in units of data warps (128 bytes). This means that if each thread in a warp requests a distinct four bytes and these 128 total bytes are contiguous and aligned to a data warp boundary, the memory controller can satisfy all requests with one transfer. This condition is called *coalescence*. This is similar to the way a CPU makes efficient use of an entire cache-line, therefore, I call each coalesced data warp a *warp-line*.

As we will see in all of my case studies, coalescence is crucial for a GPU to achieve peak throughput. So, a programmer will need to ensure that transferred data are the correct size (multiples of 128 bytes), aligned (respecting data warp boundaries), coherent (threads of a warp request distinct but contiguous data bytes), and fully used (all warp data is consumed by the threads before another warp line is transferred).

Alignment requires attention primarily for short runs, since warp lines behave like cache lines, meaning unaligned data takes one more data transfer than unaligned data, the extra transfer cost can of course be amortized across a long run of data. For example, while misaligned runs of 128 bytes take two transfers rather than one, capping coalescence efficiency at 50%, and runs of 1024 bytes take nine transfers rather than eight, capping efficiency at 89%, randomly accessing misaligned runs of 16 bytes transfer 128 bytes 7/8th of the time and 256 bytes 1/8th of the time, capping efficiency at 11.1%!

Distinctness can be relaxed, but several threads competing for the same memory address can lead to “race conditions,” where one thread overwrites a competing thread’s data. It is up to programmers to prevent race conditions in global memory by either partitioning or using costly atomics. The hardware will prevent race conditions in shared memory at the cost of serializing access for competing threads within a warp (see *bank conflicts* next).

3.4.4 Bank Conflicts

Bank conflicts occur when multiple threads within the same thread warp access the same memory bank within shared memory at the same time. The GPU hardware serializes access by competing threads to ensure correct behavior, but at the cost of reduced I/O throughput. Memory accesses involving *k* threads accessing the same bank at the same time are called *k-way bank conflicts*. When a warp has up to (and including) *k*-way conflicts, the scheduler must replay the conflicting instruction *k* times. This approach, if done consistently, reduces throughput by a factor of *k*. In the worst case, *k* can be the minimum of the number of banks or the threads per warp (both are 32 on Fermi and Kepler hardware.) Bank conflicts can be avoided if each thread in a warp accesses its own unique bank in shared memory, but this can be subtle. For example, for 64-bit data type (doubles, longlong integers) on Fermi, 32 threads each loading an element of a consecutive run results in at least a two-way bank conflict, since each loads the low order bytes from even banks before the high order bytes from odd banks[12].

[12] Kepler-class hardware has a new shared memory access mode that avoids 2-way bank conflicts for 64-bit data types.

4.0 Case Study: Memory I/O

Programmers wishing to migrate CPU code onto a GPU must learn how to do several tasks in the GPU architecture (NVIDIA, 2012, Programming Guide). This chapter demonstrates four of the most basic of these tasks:

- Write GPU data-parallel code for each thread
- Map individual threads onto data elements
- Launch thousands of parallel threads running a GPU kernel
- Set up GPU kernels from within a CPU host program

Programmers should be aware that the first two tasks occur on the GPU and can be written using enhanced CUDA C++ with differences to support data parallelism. The last two tasks occur on the CPU and can be done using normal idioms, semantics, and syntax with some differences to support launching GPU kernels with thousands of threads.

Chapter 5 introduces the concept of *Data Access Skeletons* (DASks) and demonstrates two more tasks to attain high performance:

- Write code that has efficient memory access patterns.
- Determine a balance of Instruction- and Thread-level parallelism (ILP and TLP) giving high performance.

My case study in both this chapter and Chapter 5 is the simple Copy primitive, which copies data from a source array into a separate, non-overlapping destination array. Copy is an instance of a *map pattern*, a parallel programming pattern that transforms a source array, S, of n independent data items into a separate destination array, D, of the same size, and applies a unary transform operator, $\odot$, to each data item in the input array; that is, for all i in $[0,n)$ with $s_i \in S$, compute output $d_i = \odot(s_i)$.

For copy, the parallel threads do not need to communicate or coordinate. This independence makes map patterns a good starting point to learn how to program on the GPU. Although I focus on Copy, the general map pattern can then be easily adapted for other independent operations such as Fill, Gather, or Scatter.

In this chapter, I answer the four questions in the next four sections, then show performance results for the resulting simple Copy kernel.

- *How can one setup and launch a GPU kernel?* In Section 4.1, I show how programmers can launch their GPU kernels from within a CPU host program. The CPU Host pattern sets up the template, CTA layout, and kernel parameters, all of which are necessary to launch a GPU kernel. I choose to explain how to launch a kernel before getting into the details of writing a kernel.
- *How can one launch a kernel with thousands of threads?* In section 4.2, since this is at the heart of achieving high throughput on GPUs by using massive numbers of parallel threads, I show how programmers can specify thousands of parallel threads for each kernel launch using NVidia's CTA layout parameters.
- *How can one map threads onto data elements?* In Section 4.3, I show how programmers can advantage of these thousands of parallel threads by mapping each unique thread specified by CUDA's built-in CTA parameters onto a data element.
- *How can one write GPU data parallel code?* In Section 4.4, I show how programmers can convert a CPU copy using sequential iteration into a thread-based GPU kernel using data-parallel programming. The key insight is that the hardware does the parallel scheduling, so the GPU programmer need focus on code for only a single thread.

4.1 How does one setup and launch a GPU Kernel?

Even if the program is "Hello, world," to run it on the GPU, a programmer must write CPU scaffolding code that sets up and launches the GPU kernel from the CPU host, and receives the result. So assuming that we have a simple GPU kernel for copying data in parallel (the actual kernel is in Section 4.4), how do we launch it from a serial CPU program? To launch a GPU kernel, the CPU host must allocate resources, transfer inputs onto the GPU, set up kernel CTA layout parameters, launch kernels, transfer outputs back from the GPU, and clean up allocated resources. I use a CPU host function that also includes extra validation, profiling, and debugging code in to ensure the final versions of my GPU kernels are correct, robust, and fast. Although beyond the scope of this thesis, the host function can be extended to support asynchronous streaming and launch parallel algorithms on multiple GPUs. Figure 4.1 summarizes the pattern of my CPU host functions.

CPU Host Pattern: launches GPU kernel(s) from a CPU host

Input: Varies as needed, for example: n input data items (for copy)
Output: Varies as needed, for example: n output results (for copy)

```
// CPU Host Pattern
1.  <Optional> – Choose & set up GPU device
2.  Set up CTA layout (grid of thread blocks) for each kernel
3.  Allocate memory resources on CPU & GPU
4.  <Optional> – Initialize / Set up CPU arrays
5.  Transfer inputs onto GPU from CPU
6.  Launch GPU kernel(s) with parameters (Template, CTA, Kernel)
7.  Transfer outputs from GPU onto CPU
8.  Tear down memory resources
```

Optional Variations:
1. Validation support (launch CPU method, verify CPU vs. GPU results)
2. Profiling support (create/destroy timers, start/stop timers, output performance results)
3. Debugging support (create/transfer intermediate arrays, add verification methods, etc.)
4. Asynchronous streaming (create/destroy streams, overlap transfer/computation).
5. Multiple GPU support (partition inputs across GPUs, merge outputs from GPUs).

Figure 4.1: A high level overview of CPU Host pattern.

Figure 4.2 presents a specific instance of the CPU host pattern using simplified source code from a CPU host function that wraps one of my GPU Copy kernels (shown later in Section 4.4).

```
CPU_Host_Copy( h_input, h_output, nElems )
   // 1. Optionally - Choose GPU device
   ...
   // 2. Set up Launch Parameters (Template, CTA, and Kernel)
   typedef U32 valT;      // Underlying ‘value’ type
   U32 nWork       = 4u;  // 4, Work items per-thread
   U32 logBankSize = 5u;  // log<2>(32), Channels per-bank
   U32 logWarpSize = 5u;  // log<2>(32), Threads per-warp
   U32 BlockSize =  64u;  // 64, Threads per-block
   U32 GridSize  = (nElems+(BlockSize-1))/BlockSize;  // Vary 1D grid with n
   dim3 Block(BlockSize, 1, 1); // Block Layout (1D shape & fixed size)
   dim3 Grid(GridSize,  1, 1);  // Grid Layout  (1D shape & varying size)
   // 3. Allocate Memory on GPU
   U32 mem_size_values = nElems * sizeof(valT);
   valT * d_input  = nullptr;
   valT * d_output = nullptr;
   cudaMalloc( (void**)&d_input, mem_size_values );
   cudaMalloc( (void**)&d_output, mem_size_values );
   // 4. Optionally – Setup Host input arrays
   ...
   // 5. Copy 'inputs' onto GPU
   cudaMemcpy( d_input, h_input,
               mem_size_values, cudaMemcpyHostToDevice );
   // 6. Launch Copy Kernel
   HC_Copy_RowByRow                C
   <
      // C++ Template parameters
      valT, logWarpSize, logBankSize, BlockSize, nWork        T
   >
   <<<
      // CTA Layout parameters (Grid of Thread Blocks)        L
      Grid, Block
   >>>
   (
      // Kernel parameters                                    K
      d_output, d_input, 0, nElems - 1
   );
   // 7. Copy 'outputs' from GPU to CPU
   cudaMemcpy( (void *)h_output, (void *)d_output,
               mem_size_values, cudaMemcpyDeviceToHost );
   // 8. Cleanup resources
   cudaFree( d_output );
   cudaFree( d_input );
end CPU_Host_COPY
```

Figure 4.2: Example usage of the *CPU Host* pattern to launch a GPU *Copy* kernel.

As is apparent in Figure 4.2, the code that launches GPU kernels from a CPU host can be confusing because the programmer must work with up to three separate sets of parameters:

- Kernel parameters, which are the normal function parameters passed into the code, like the number of data elements *n*

- CTA layout parameters, which structure the threads into blocks and grids. This will be explained in more detail in Section 4.2.
- Optional C++ template parameters, which simplify the setup of optimization experiments.

I first specify these template parameters in C++ using single angle brackets (< … >). Next, the CTA layout parameters are used to structure the parallel threads as a grid of thread blocks for each GPU kernel. The CTA layout parameters are specified in CUDA using triple angle-brackets (<<< … >>>). (I will discuss in Section 4.2.1 how to choose appropriate CTA layout parameters up front.) Finally, the normal kernel parameters passed into the GPU kernel are specified in C++, as in most programming languages, using single parentheses.

Figure 4.2 shows, as a specific example, the parameters for my GPU copy invocation. The glyph Ⓒ marks where the launch code for the copy kernel in this example starts. The glyphs ◆T ◆L ◆K respectively represent the three sets of parameters: template, CTA, and kernel. (I have spread the three sets of parameters across multiple lines to make them more visible. Most programmers would use a more terse syntax.) As shown in Figure 4.2, the CPU host pattern mostly sets up and cleans up necessary resources around a set of GPU kernel launches. The CTA layout parameters structure thread parallelism and are unique to GPU data-parallel programming. Section 4.2, which follows, goes into greater detail about how to structure the thread hierarchy in CUDA for each GPU kernel using the 2-level CTA layout parameters.

4.2 How does one launch a kernel with thousands of threads?

One obstacle to programming a GPU kernel is the need to specify up front a thread layout structure, the cooperative thread array (CTA), which the GPU hardware uses to schedule the tens of thousands of threads onto hundreds of processing cores. Each kernel, no matter how simple, requires a CTA layout, but choosing good sizes and shapes for the CTA layout requires the programmer to develop some intuition for what thread layouts work well, and optimal sizes and shapes require many experiments within the specific application. Here are the technical details for specifying layouts.

Recall from Section 3.3 that threads on a GPU are structured in a four-level hierarchy. Each individual thread belongs to a thread warp, which belongs to a thread block, which belongs to a grid. In order to launch kernels with thousands of threads, NVidia requires that programmers must specify, at GPU kernel launch, two triples ‹*x*,*y*,*z*› the 2D grid of blocks and a 3D array of threads in the blocks. To aid my own understanding, I use a 6-tuple ‹*gw*, *gh*, 1, *bw*, *bh*, *bl*› to represent the grid and block layout parameters as a single unit. However, CUDA actually uses a triple angle-bracket syntax (`<<<grid, block, …>>>`). I also use the 6-tuple ‹*bx*, *by*, *bz*, *tx*, *ty*, *tz*› to indicate the unique thread location of each individual thread within the CTA layout.

To specify the CTA layout up front, the GPU programmer must plan how to partition the data and how to map individual threads onto the data elements. The programmer can simply specify a 1D grid and a 1D thread block layout, with two CTA layout parameters ‹*gw*,1,1,*bw*,1,1›, but the option of having multiple CTA layout parameters allows programmers the flexibility to treat the underlying data as 1D, 2D or 3D, as appropriate for their problem space. Most of the GPU kernels in my case studies are 1D, but large, so I typically use 2-3 CTA parameters per kernel as a 1D or 2D grid and a 1D block as ‹*gw*,*gh*,1,*bw*,1,1›.

The code in Figure 4.3 shows a 2-3 parameter CTA layout for a 1D data set [0,*n*).

```
template<TBS, gridRS, nWork> // Input Size, Row Size(m*c),
                                Work per-thread
Layout_1D( n, Grid, Block )        // Input Size, Grid layout, Block Layout
  DBS = nWork*TBS;              // Fixed, Data Block Size
  nBlocks = ⌈n/DBS⌉;            // Varies, Cover data with data blocks
  if (nBlocks <= 65534)        // 1D or 2D Grid Layout?
    gridW = nBlocks;
    gridH = 1;
  else
    mSQ = ⌈√nBlocks⌉;          // Start with a square layout
    gridW = ⌈mSQ/gridRS⌉·gridRS;  // nCols is a multiple of 'Row Size' Hint
    gridH = ⌈nBlocks/gridW⌉;       // nRows needs to cover data
  end if
  Block = dim3( TBS, 1, 1 );        // Block layout (1D)
  Grid  = dim3( gridW, gridH, 1 ); // Grid layout (2D or 1D)
end Layout_1D

// Example Usage
...
TBS    = 128;                   // 128, Pick a fixed 1D thread block layout
nWork  = 4;                     //   4, Amortize costs across work-items
nSMs   = 14;                    //  14, number of SMX's on a GTX Titan
nConBlocks = 16;                //  16, expected number of concurrent blocks per SMX
gridRS = nSMs * nConBlocks;     // 224, A good starting row size for my grid
Layout_1D<TBS,gridRS,nWork>( n, Grid, Block ); // Compute my CTA Layout
```

Figure 4.3: *Compute a CTA layout.* In this example, I pick a fixed block size and a fixed number of columns per grid and then allow the number of rows per grid to vary as needed to cover the data.

As shown in Figure 4.3, the user picks a fixed-size thread-block size (TBS) and a fixed size amount of work per-thread (*nWork*) from which the code computes a fixed-size data block size ($DBS = nWork{\cdot}TBS$). From the fixed size data block size, I compute the number of data blocks ($m=\lceil n/DBS \rceil$) needed to fully cover the data set [0,*n*). Fully covering the data with data blocks implies that the last data block may only be partially full and thus my GPU kernels will require range checking to avoid data access errors. Initially, I used a one-to-one mapping between thread and data blocks. Recall that Fermi architectures limit their grid dimension values to the range $[1,2^{16})$. As a result, my layout function determines if a 1D or 2D grid layout is needed to cover all the data blocks (test: $m \geq 65{,}534$). If a 2D grid layout is needed, I start with the square root of *m* as a good first approximation ($\sqrt{m} \times \sqrt{m}$). I then modify that number by a fixed-size row hint (*gridRS*) to get the final 2D layout (*rows*×*cols*), which creates a 2D layout of data blocks needed to fully cover all the data. Unfortunately, computing the 2D grid layout as described usually results in some over-coverage because the last data row is usually only partially full (implying that there are some thread blocks where the corresponding fixed-size data blocks are completely out of range). Thus range checking in my kernels is required to prevent data access errors.

CTA Layout Guidelines: Using fewer CTA layout parameters results in fewer mapping operations (as will be shown later in Figure 4.6.) Consequently, I prefer 1D layouts over 2D layouts, and 2D layouts over 3D layouts. Since thread scheduling on each SM multi-core is warp-based, I also prefer that the thread block width (*bw*) parameter be a multiple of the *WarpSize* (32), in order to fully utilize all the SP processing cores on each SM. To support the multi-issue GPU hardware capability, I prefer to have at least two or four thread warps per-thread block (64 or 128 threads per thread-block). I generally prefer fixed sized constants for most of my CTA parameters—the only exception being the one that I allow to vary with input size (either the grid rows or grid columns -- *gw*, *gh*). I recommend including the fixed-size CTA parameters as part of the kernel's C++ template parameters. This approach supports experimentation while still allowing the GPU kernels to know the actual CTA layout at compile time for better performance.

4.3 How does one map threads onto data?

To process all data, GPU programmers must map individual threads onto unique data locations in the data set. In addition, all threads collectively must fully cover all data in the data set. Therefore, the programmer must combine these CTA layout parameters (2-5 variables) with the current thread's unique identifying parameters in order to map the thread's access request onto a single data location.

To support mapping at runtime, CUDA provides four CTA layout parameters. Each of these four parameters are global[1], built-in, and read-only 3-tuples as ‹*x*,*y*,*z*›. They are named `gridDim`, `blockDim`, `blockIdx`, and `threadIdx` respectively. The `gridDim` and `blockDim` layout tuples refer to the original CTA layout parameters. (This means that `gridDim.`‹*xyz*› and `blockDim`.‹*xyz*› together are equivalent to ‹*gw*, *gh*, 1, *bw*, *bh*, *bl*› in my nomenclature). The `threadIdx` and `blockIdx` tuples uniquely identify the current running thread within the current thread block within the grid. (Together they are equivalent to ‹*bx*, *by*, 1, *tx*, *ty*, *tz*›, in my nomenclature.) Each `threadIdx` and `blockIdx` parameter value ‹*x*,*y*,*z*› is upper-bounded by a corresponding `blockDim` and `gridDim` value. For example, given `threadIdx.x` (*tx*) and `blockDim.x` (*bw*), then *tx* will be bounded above by *bw*, i.e. $tx \in [0, bx)$. All four built-in CTA layout parameters are therefore used to map individual threads onto their corresponding data items. This approach is similar to mapping multi-dimensional arrays onto 1D memory layouts. Mapping the `blockIdx` or `threadIdx` onto a single unique block or thread ID (*bid* or *tid*) involves computing 3D slabs, 2D rows, and 1D columns and adding them all up.

Table 4.1 shows code snippets that can be used to map the CTA layout variables onto a single per-thread data location using a 1D or a 2D grid vs. a 1D, 2D or 3D block layout.

[1] This means that these four CTA layout parameters can always be accessed and used at anytime from anywhere within kernel code and invoked function code without any need to pass them through as kernel or function parameters.

	1D Block <*bw*,1,1>	**2D Block** <*bw*,*bh*,1>	**3D Block** <*bw*,*bh*,*bl*>

1D Grid <*gw*,1,1>	`bid=blockIdx.x;`
	`tid=threadIdx.x;`
	`TBS=blockDim.x;`
	`DBS=WPT*TBS;`
	`dataOff=(bid*DBS)`
	`+tid;`
		`bid=blockIdx.x;`
		`tX=threadIdx.x;`
		`tY=threadIdx.y;`
		`bW=blockDim.x;`
		`bH=blockDim.y;`
		`tid=(bW*tY)+tX;`
		`TBS=bW*bH;`
		`DBS=WPT*TBS;`
		`dataOff=(bid*DBS)`
		`+tid;`
			`bid=blockIdx.x;`
			`tX=threadIdx.x;`
			`tY=threadIdx.y;`
			`tZ=threadIdx.z;`
			`bW=blockDim.x;`
			`bH=blockDim.y;`
			`bL=blockDim.z;`
			`slab=bW*bH;`
			`tid=(slab*tZ)+`
			`(bW*tY)+tX;`
			`TBS=slab*bL;`
			`DBS=WPT*TBS;`
			`dataOff=(bid*DBS)`
			`+tid;`

Runtime Ops	5	9	13
Template Ops	3	5	7

2D Grid <*gw*,*gh*,1>	`bX=blockIdx.x;`
	`bY=blockIdx.y;`
	`gW=gridDim.x;`
	`bid=(gW*bY)+bX;`
	`tid=threadIdx.x;`
	`TBS=blockDim.x;`
	`DBS=WPT*TBS;`
	`dataOff=(bid*DBS)`
	`+tid;`
		`bX=blockIdx.x;`
		`bY=blockIdx.y;`
		`gW=gridDim.x;`
		`bid=(gW*bY)+bX;`
		`tX=threadIdx.x;`
		`tY=threadIdx.y;`
		`bW=blockDim.x;`
		`bH=blockDim.y;`
		`tid=(bW*tY)+tX;`
		`TBS=bW*bH;`
		`DBS=WPT*TBS;`
		`dataOff=(bid*DBS)`
		`+tid;`
			`bX=blockIdx.x;`
			`bY=blockIdx.y;`
			`gW=gridDim.x;`
			`bid=(gW*bY)+bX;`
			`tX=threadIdx.x;`
			`tY=threadIdx.y;`
			`tZ=threadIdx.z;`
			`bW=blockDim.x;`
			`bH=blockDim.y;`
			`bL=blockDim.z;`
			`slab=bW*bH;`
			`tid=(slab*tZ)+`
			`(bW*tY)+tX;`
			`TBS=Slab*BL;`
			`DS=WPT*TBS;`
			`dataOff=(bid*DBS)`
			`+tid;`

Runtime Ops	8	12	16
Template Ops	5	7	9

Table 4.1: This table shows code snippets for mapping 2-5 CTA parameters down onto a single per-thread data location used to access data in global memory. Grid dimensions (1D & 2D) and block dimensions (1D, 2D, and 3D) are on the vertical and horizontal axes respectively. The lighter grey color indicates lines of code that get compiled away as constants when using C++ template parameters to redundantly pass in the corresponding fixed-size CTA layout parameters.

The mapping between threads and data can be simple and straightforward, but because the current CUDA documentation does not actually show how to do this mapping, I have included my code for six different block and grid dimensions in Table 4.1. (There are many other possible ways to map these variables.) My mapping for each of the six code snippets is computed in four steps:

1. The block ID (*bid*) for a thread block within a grid is first computed from the `blockIdx` and `gridDim` parameters.
2. The thread ID (*tid*) for a thread within a thread block is then computed from the `threadIdx` and `blockDim` parameters.
3. Next, the fixed-size DBS can be computed from fixed-size C++ template parameters (*nWork* and TBS) or from the `blockDim` parameter.
4. Finally, the per-thread data location is computed from the *bid*, *tid* and DBS.

Also included in Figure 4.4 are my best estimates of how many operations CUDA takes to perform each mapping via the corresponding code snippet. As you can see from the code, the more CTA parameters involved in the mapping, the more operations it takes. The "Runtime Ops" rows show the total number of operations required to perform the mapping when it directly accesses the CUDA layout variables. In contrast, the "Template Ops" rows show the reduced number of operations expected when redundantly passing some of the CUDA layout parameters as fixed-size C++ template parameters into the kernels. In Figure 4.4, I have also grayed out the lines of code that the compiler elides when using C++ template variables.

Each mapping code snippet consists of simple loads, multiplies, and adds. Any required CUDA CTA layout parameters (`gridDim`, `blockIdx`, `blockDim`, `threadIdx`) must first be loaded from special constant read-only registers into normal registers before they can be used. For Fermi and CUDA architectures, the CUDA platform compiler will often replace a multiply operation that is followed by an add operation with a single fused-multiply-and-add (FMA). In

The code snippets in Figure 4.4 have many read-after-write (RAW) data dependencies, which reduce opportunities for the SM warp schedulers to hide stalls. For fewer instructions and stalls, GPU programmers should prefer 1D over 2D mappings and 2D over 3D mappings.

4.4 How can one write GPU data parallel code?

Now that we have some initial ideas about how to setup and launch a GPU kernel from a CPU host, and how to launch a kernel with thousands of parallel threads, and how to map those threads onto data elements, it's time to actually drill down on the task of writing a GPU kernel that supports data-level parallelism.

For a serial solution, many programmers, when given a large data set that needs to be processed, naturally draw upon the "iteration pattern" from their programmer's tool box. A nested loop structure that iterates over all data values easily processes all data in a large dataset. The loop instructions therefore simplify a programmer's life by reducing the amount of code that needs to be written while also dynamically supporting any input size (*n*). However, as useful as loop instructions are in increasing programmer productivity, what is really important are the actual transform instructions that convert a single input into a single output in order to solve the problem at hand. These important transform instructions are typically found as the innermost block of code within the nested looping structure.

For a parallel solution, such as on a multi-computer, one natural way to solve these problems is to parallelize the loops themselves, creating one thread per loop iteration to process the data from each iteration. Such an approach lets the machine handle scheduling of iteration instances onto threads and then onto cores. With *n* iterations, *n* data items can be covered. Of course, for a multi-computer there must be some way to specify parallelism. Parallelism is often specified in parallel programming platforms, like MPI or Clik, using a `parallel_for` instruction. In these instances, the framework creates parallel threads, schedules threads, maps data items onto threads, executes threads, and retires threads. Given these advantages for typical multi-computer parallel solutions, I would rewrite my loops as parallel looping structures.

For a GPU specific solution, the CUDA framework supports a data parallel view of data with one thread per data item. However, the nested loop structures in software are replaced by a two-level cooperative thread array (CTA), which groups threads into thread blocks and thread blocks into a grid. The hardware giga-thread scheduler schedules thread blocks onto SMs, and each SM warp scheduler schedules thread warps onto SPs. Since looping has effectively been moved out of software and onto the hardware schedulers, each thread needs to uniquely identify itself within the CTA hierarchy in order to know which data item to work on. As before, the same transform instructions that I would loop over in my serial solution becomes the core of my GPU data parallel solution.

The CUDA programming platform enhances the C++ language with a few extensions to express and support data parallel programming. Each GPU kernel is written from the point of view of a single generic thread (within the CTA). Each GPU data parallel kernel then typically consists of the following six steps:

Step 1: *Mapping Data*: During this step, programmers write code that uniquely identifies the current thread and maps the current thread ID onto a specific data location (input and output). This important step was discussed previously in Section 4.3.

Step 2: *Setting Up*: An optional step where programmers write code that can allocate, load, or create any necessary variables to support algorithm state and context.

Step 3: *Loading Input*: During this step, programmers write code that loads a single data item as input from the computed input location. Range checks may be required to prevent out-of-range memory accesses.

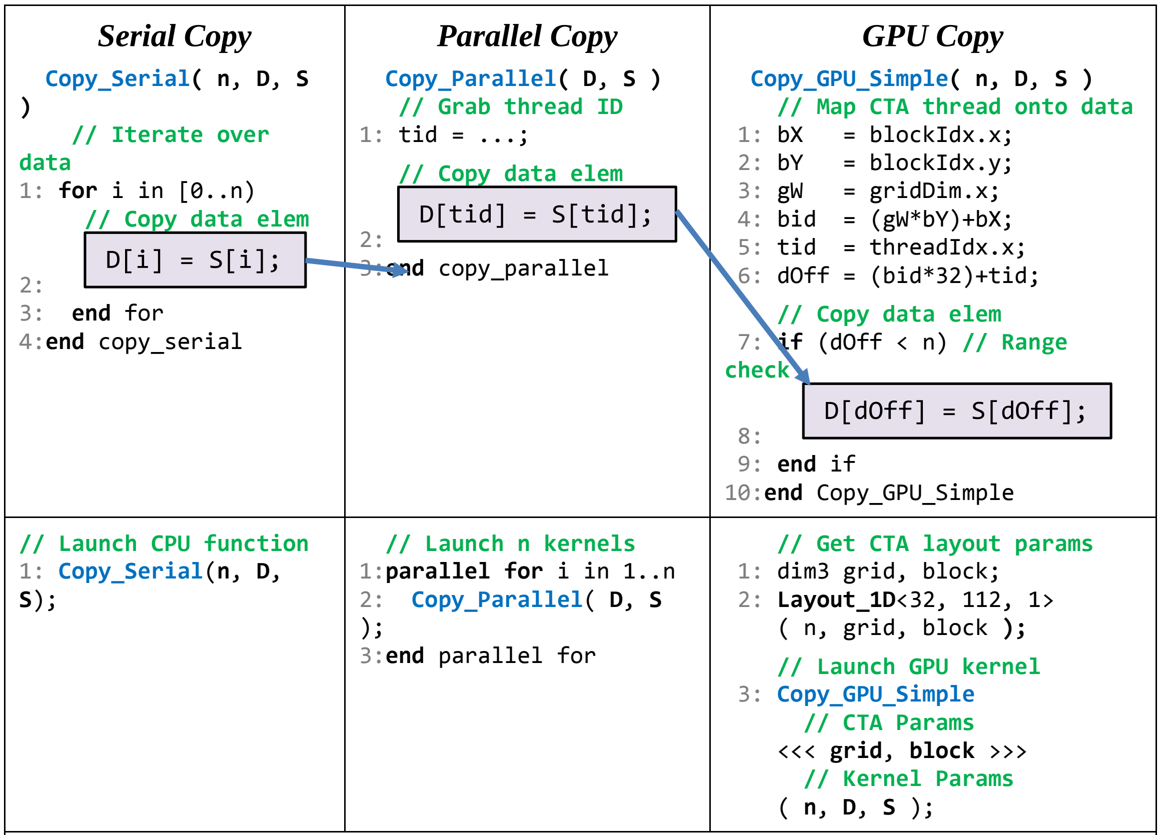
Step 4: *Transforming Data*: During this important step, programmers write code that transforms the input data item into an output result.

Step 5: *Storing Output*: During this step, programmers write code that stores the result item as output to the computed output location. Again, range checks may be required.

Step 6: *Cleaning Up*: An optional step where programmers write code to clean up any allocated resources from the "Setting Up" step.

The "Transforming Data" step (Step 4) is the important payoff step where the original problem gets solved. Just as the loop instructions are necessary and useful overhead for solving a large problem in a serial environment, the other steps above (1-3, 5-6) are also necessary and useful overhead to solve a large massively multi-threaded problem in a GPU data parallel environment. I call the important transforming data step the "Body," and the rest of the supporting framework steps the "Skeleton" of the GPU kernel.

I just described three different approaches for transforming input into output. The CPU serial looping, multi-computer loop parallel, and GPU data parallel versions of Copy are all shown as pseudo-code in Figure 4.4.



Serial Copy

```
  Copy_Serial( n, D, S
)
    // Iterate over
data
1: for i in [0..n)
      // Copy data elem
      D[i] = S[i];
2:
3:   end for
4:end copy_serial
```

Parallel Copy

```
  Copy_Parallel( D, S )
    // Grab thread ID
1: tid = ...;
    // Copy data elem
    D[tid] = S[tid];
2:
3:end copy_parallel
```

GPU Copy

```
  Copy_GPU_Simple( n, D, S )
     // Map CTA thread onto data
 1: bX   = blockIdx.x;
 2: bY   = blockIdx.y;
 3: gW   = gridDim.x;
 4: bid  = (gW*bY)+bX;
 5: tid  = threadIdx.x;
 6: dOff = (bid*32)+tid;
     // Copy data elem
 7: if (dOff < n) // Range
check
       D[dOff] = S[dOff];
 8:
 9: end if
10:end Copy_GPU_Simple
```

```
// Launch CPU function
1: Copy_Serial(n, D,
S);
```

```
  // Launch n kernels
1:parallel for i in 1..n
2:  Copy_Parallel( D, S
);
3:end parallel for
```

```
   // Get CTA layout params
1: dim3 grid, block;
2: Layout_1D<32, 112, 1>
   ( n, grid, block );
   // Launch GPU kernel
3: Copy_GPU_Simple
     // CTA Params
   <<< grid, block >>>
     // Kernel Params
   ( n, D, S );
```

Figure 4.4: *Top Row, Left Panel*: A simple serial CPU program to copy *n* elements from source to destination using a loop. *Top Row, Middle Panel*: An equivalent data-parallel kernel would copy *n* elements using *n* threads. *Top Row, Right Panel*: An actual GPU kernel instance to copy *n* elements using *n* threads. *Bottom Row* (all three panels): Examples of how to invoke the function and kernels. With all three functions, the shaded boxes contain the useful payoff code (*Body*) that the GPU programmer implements (in this case: copying). The rest of the code is the necessary framework code (*Skeleton*) in order to iterate over *n* elements or process *n* elements in parallel.

As can be seen in Figure 4.4, all three versions have their assignment operations boxed because they provide us with our first examples of DASks. By replacing the "Body" code inside the shaded boxes at source line "2" or "8," a number of programs based on the map pattern can be generated from the above "Skeletons", such as Copy, Fill, Count, and Find. For Fill, a programmer could replace the copy statement at source line 2 (or 8) by a fill statement `D[tid] = fillVal;`.

As also can be seen from Figure 4.5 (left vs. right panel), the loop overhead of the CPU serial function has been replaced by the CTA layout and mapping overhead of the GPU data-parallel kernel. The resulting parallel GPU copy is conceptually the same as the parallel multi-computer kernel in Figure 4.5; however, the extra complexity comes from having to map the 2-level CTA layout kernel parameters that represent a single thread down onto a single data location. This simple GPU copy works correctly for valid input, but its performance runs at only ~25% of peak I/O throughput for large input sizes (see Figure 4.7).

To support massive parallelism via threads, GPU programmers also must make some up-front decisions about the CTA layout before coding a simple GPU kernel such as Copy. To do this, GPU programmers must first set up the initial thread structure using the CTA layout parameters (grid, block). This setup appears in Figure 4.1 (bottom row, right panel, source line 3) as an extra line of kernel launch parameters (`<<< grid, block >>>`). Later, when they are coding the GPU kernel, GPU programmers must map the CTA layout (as four built-in kernel parameters) for each individual thread down onto a single data location. This mapping code shows up in Figure 4.1 (top row, right panel) as source lines 1-6. I present more details on this two-step process for launching parallel kernels and mapping threads onto data in Section 4.3.

As will be seen in Chapter 5, I will present three DASks for efficiently accessing memory. I developed these DASks with two main goals: to help hide the complexities of GPU programming so that the GPU programmer can focus on solving their problems via their algorithms; and to achieve solid parallel performance on GPU hardware.

Otherwise, GPU programmers must expend a great deal of effort to learn the complexities of the GPU hardware that can help or hinder performance. For example, in order to increase performance via TLP, GPU programmers need to be aware of the underlying hierarchical thread structure in hardware. I initially suggested that GPU programmers should think about their parallel programs from the point of view of a single thread. It is vital that they should not lose sight of the underlying four-level hierarchical thread structure: A thread belongs to a thread warp, which belongs to a thread block, which belongs to a grid. When executing a kernel, the GPU schedules thread blocks within each grid onto SMs. Each SM’s warp-scheduler then schedules concurrently running thread warps from thread blocks onto the SPs within each SM.

As each thread block completes execution, the GPU schedules additional thread blocks onto the SMs until all thread blocks within the grid have been processed.

To support data parallelism in my GPU programs, I at one time wrote my own GPU kernels in ten rough stages as follows:

1. *Understand Structure*: Conceptually, I would look at the actual serial algorithm to adapt (or I would think of how could write this kernel as a serial program using some looping structure that iterates over the *n* elements in the input data set).
2. *Find "Body"*: I found (or thought about) the innermost code in the actual (or conceptualized) looping structure. This is the important code that transforms one input element into one output result. This innermost transform code forms the core of my GPU kernel.
3. *Write Core*: Based on the innermost transform code, I wrote the core of GPU kernel that loads, transforms, and stores a single data work-item.
4. *Map Threads*: I added code that maps each individual thread into a unique data work-item. I also usually had to go back and add range checks around any load or store instructions to prevent access errors.
5. *Serial Test*: I verified my kernel with exactly one thread on one data item.
6. *Increase parallelism to one Thread Warp*: I verified that my code works for all threads within a single warp. Meaning, I set my CTA layout to a single thread warp (32 threads) and tested my code on a single data warp (32 data items).
7. *Coordinate Threads*: I considered how the threads within a thread block might need to communicate and coordinate with other threads to ensure correct parallel behavior. I shared data across threads using arrays in shared memory. I also might need to insert barriers instructions or use atomics to ensure that all threads

are reading or writing correct data without conflicts. I then rewrote my code to support the desired communication or coordination across threads.

8. *Increase parallelism to one Thread Block*: I verified that my code works for all threads within a single thread block (128 threads per thread block is a good starting case) on a single data block (128 data elements).
9. *Increase parallelism to an entire Grid of thread blocks*: I verified that my code works correctly for all threads in a large grid of thread blocks.
10. *Complete Algorithm*: The kernel should now be working correctly. If this kernel is just one stage of a larger algorithm, I also needed to verify that this stage interacts correctly with the other stages preceding and following it.

However, after inventing my various Data Access Skeletons (DASks), I now do things differently. I take one of my working DASks (as discussed in Chapter 5) that best fits my problem and replace the "Body" sections (shaded boxes) with my desired transform data code and then start verifying. Of course, I still need to think about how to communicate and coordinate partial results across threads within a thread block and between kernels in a multi-kernel parallel algorithm. However, the main result of jump starting my kernel development with my DASks is that I am more productive.

4.5 Simple Copy Results

Figure 4.5 shows the initial throughput (in gigabytes per second) for increasing input sizes (*n*) of my simple Copy kernel (code shown in Figure 4.4, right panel).

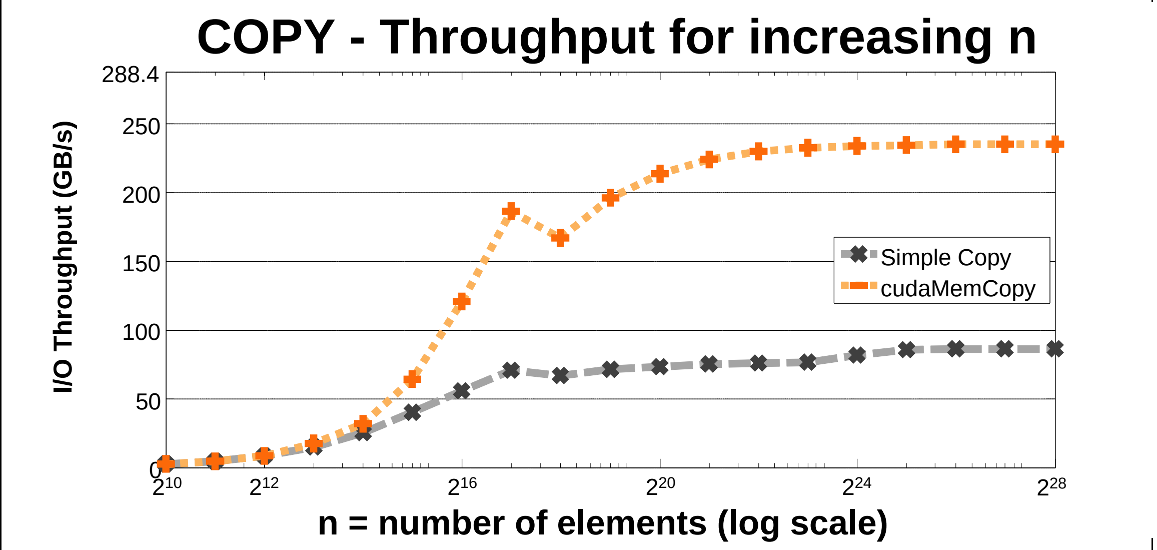


Figure 4.5: Simple throughput for increasing *n*. All tests run on a GTX Titan.

This simple kernel works correctly but disappointingly achieves only about 25% of maximum peak throughput available on the GTX Titan and is about 2.7× slower than the built-in CUDA library function `cudaMemCopy`. As we will see in Chapter 5, this is because this simple kernel does not take full advantage of either TLP or ILP.

4.6 Conclusion

In this chapter, I have introduced the four main tasks—writing CPU host functions to launch GPU kernels, launching thousands of parallel threads using the CTA layout, mapping each parallel thread within the CTA onto a data work-item, and writing thread centric code for data parallel kernels—that GPU programmers should learn in order to get a GPU kernel up and running. I demonstrated these four tasks on a simple Copy primitive. Even though the resulting kernel works correctly, it has disappointing performance (only 25% of peak throughput).

To improve performance, in Chapter 5, I will introduce three *data access skeletons* (DASks) and two *block access skeletons* (BASks). These skeletal frameworks provide three main benefits:

1) They support efficient access patterns into memory

2) They support experiments on balancing ILP and TLP to find the best throughput performance.

3) They provide an already working framework that hides much of the complexity of writing GPU kernels with high performance.

My DASk and BASk kernel frameworks allow me to get a jump start on writing new GPU code which makes me more productive. The underlying framework code forming the skeletons has already been tested, works correctly, and achieves high throughput. Each DASk contains replaceable “body” sections (that I show throughout this thesis as shaded boxes) that are intended to be replaced with new code to solve new problems. So, the programmer can replace these “body” sections with their own code to solve their own problem with high confidence that the resulting kernels are close to working correctly. Of course, this approach doesn’t solve everything, the programmer still needs to think about how to structure their CTA layout on launch, and how to best write their code to best deal with GPU hardware issues. To that end, my case studies will demonstrate various ways that I have dealt with those issues using my DASks and BASks.

5.0 Data Access Skeletons (DASks)

In this chapter, I continue my case study using the Copy primitive to learn how to aim for high performance on the GPU. In Chapter 4, the focus was on coding a simple Copy that works correctly on a GPU by learning four main tasks: launch a GPU kernel from a CPU host; launch thousands of parallel threads for each kernel; map each thread onto a data work-item; and write GPU thread-centric data-parallel code;. The final kernel was correct but had poor throughput as compared to the peak throughput available on cards like the GTX 580 and GTX Titan.

In this chapter, I focus on improving the Copy primitive's performance on the GPU by introducing two additional tasks.

1) Write code that has efficient memory access patterns.
2) Determine a balance of Instruction-level and Thread-level parallelism (ILP and TLP).

To help support both performance tasks, I introduce three *Data Access Skeletons* (DASks)--: *Block*, *Column*, and *Row*. And two *Block Access Skeletons* (BASks) – *Block by Block* and *Warp by Warp*. In addition to supporting efficient memory access patterns and supporting ILP and TLP, these skeletal frameworks provide also provide an already working framework that helps hide much of the complexity of writing GPU kernels with high performance.

The underlying framework code forming these skeletons has already been tested, works correctly, and achieves high throughput. Each DASk contains replaceable "body" sections (that I show throughout this thesis as shaded boxes) that are intended to be replaced with new code to solve new problems. So, the programmer can replace these "body" sections with their own code to solve their own problem with high confidence that the resulting kernels are close to working correctly.

To support my DASks and BASks in the context of actual algorithms for my case studies, I implement my GPU kernels using C++ templates for generic programming and better performance. Generic programming[1] (Alexandrescu, 2001) helps support many data types with only one function, helps to eliminate hard-coded magic numbers from inside my kernels, and, most importantly, permits me to try many different CTA layout configurations without the need to tweak the underlying GPU kernel code for each new configuration. Better performance occurs because many C++ template parameters are treated as hard-coded constants that the compiler can use to optimize instructions (or elide away) at compile time instead of runtime.

Returning to the three types of DASks, The *Block* DASk is a generalization of the simple GPU copy (see Figure 4.1, right panel, in Chapter 4) and supports both ILP and TLP for better overall performance. The *Column* DASk assumes a 1D block and a 2D grid with a fixed number of grid columns where the number of grid rows varies dynamically with the input size. The *Row* DASk also assumes a 1D block and 2D grid where with the number of rows are fixed and the number of columns are allowed to vary dynamically with the input size.

All three DASks support experiments on both ILP and TLP via C++ template parameters[2] ‹*nWarps*, *nWork*›. The parameters exist in each DASk, the programmer should take advantage of them. The *nWarps* parameter allows the programmer to experiment with TLP by varying the number of threads per thread block. The *nWarps* parameter is actually redundant information to the existing CTA layout but by making it a compiler constant, performance can be increased in the kernel code. The *nWork* parameter allows the programmer to experiment with ILP by varying the number of data elements (work-items) per thread in their code. Even though the parameter is there, the programmer must write code in their “body” sections to support up to k work-items and then set the nWork parameter to the desired amount of work per-thread.

[1] As described in the book “Modern C++ Design” (Alexandrescu, 2001).
[2] As described in the book “The C++ Programming Language, 4th edition” (Stroustrup, 2013).

To whet your appetite for the results of using these skeletons, here are the throughputs as a function of input size (n) for my copy implementations of each of my three data access skeletons (Block, Column, Row), along with results of a simple copy implementation and NVidia's built-in copy primitive, `cudaMemCopy`. The simple implementation falls behind at $n = 2^{14}$, but all the rest are competitive with NVidia's `cudaMemCopy`, as seen in Figure 5.1. (So if you are just copying plain old data types, use `cudaMemCopy`.)

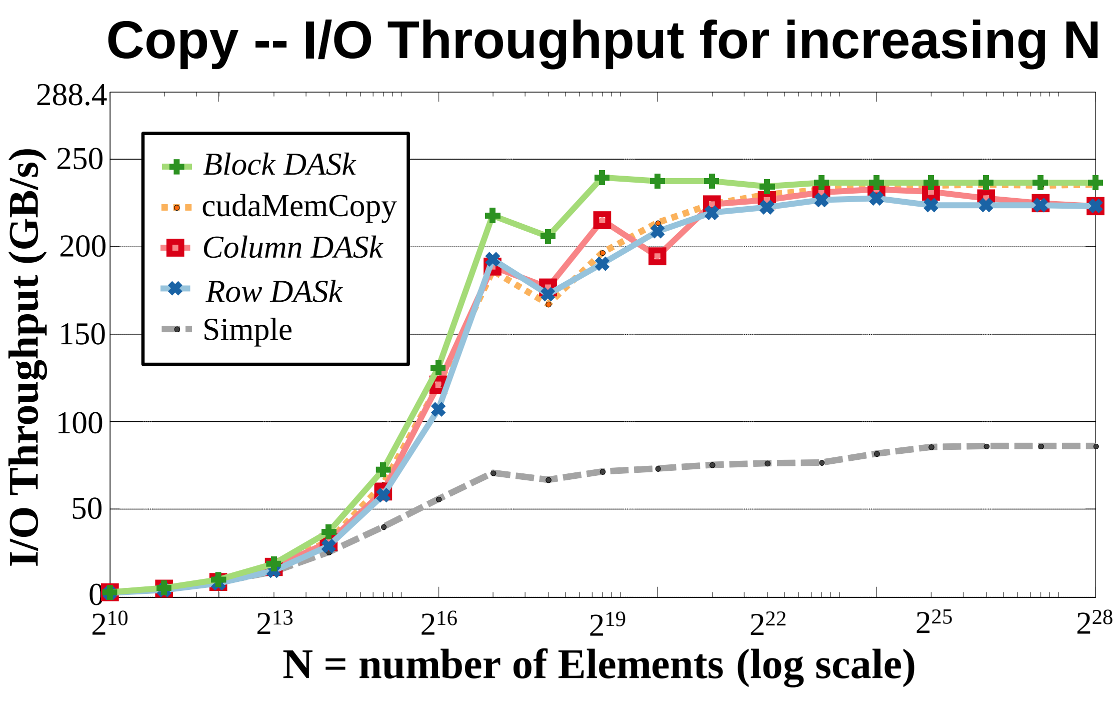
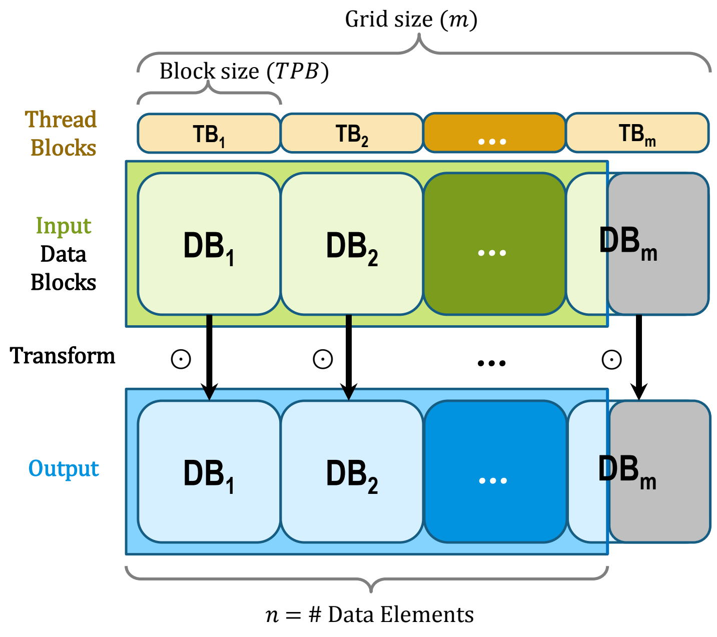


Figure 5.1: *Copy Performance* -- I/O Throughput (y-axis) for increasing n (x-axis, log-scale) for my three *Data Access Skeletons* (*Block*, *Column*, and *Row*). Performance of a simple (one thread per data item) copy degrades around $N=2^{15}$, but **`cudaMemCopy`** method can be used over the entire range.

5.1 *Block* DASk

The *Block* DASk (see Figure 5.2) is based on the simple Copy kernel (see Figure 4.1, right panel in Chapter 4) but generalized with additional C++ template parameters that support performance experiments on both ILP and TLP. For ILP, I use software pipelining on multiple work items (data elements) per-thread per data block, the amount of work per thread is specified via a work-per-thread (*nWork*) C++ template parameter. For TLP, I support massive thread parallelism via a dynamic CTA layout for fixed 1D block sizes and for 2D grids (meaning, fixed columns and varying rows), the static CTA layout parameters are also specified as C++ template parameters: threads per thread block (*TBS*) and block columns per grid (*GridCols*). By turning the Block DASk into a C++ template kernel, a programmer can experiment with different configurations (work per thread, CTA layout) by simply changing template parameters instead of tweaking hard-coded magic numbers. With the Block DASk, the data is divided into *m* fixed size blocks and organized into a 2D or 1D grid, depending on the number of thread blocks needed to fully cover all the data (using the `Layout_1D` function, shown in Figure 4.5 in Chapter 4).



Computations:
n = Number of data elements
$nWork$ = Pick *Work per-thread*
TBS = Pick *Thread Block Size*

CTA Layout:
L1: Compute Data Block Size
DBS = *nWork***TBS*
L2: Compute 1D Grid Size
m = Data blocks per-grid
$m = \lceil n/DBS \rceil$

Map: thread block onto data block
bid = (*grid.y***grid.w*) + *grid.x*
tid = *block.x*
dataOffset = (*bid***DBS*)+*tid*

Figure 5.2: The *Block* DASk. 1) Pick a fixed-size thread block, fixed-size work per-thread (*nWork*), and compute the corresponding fixed-size data block. 2) Partition the data array into *m* fixed-size data blocks that fully cover the data range [0, *n*). 3) Launch one thread block per data block. 4) Inside each kernel, map each thread block onto its corresponding data block. 5) Each thread within the thread block processes its assigned work (transforms ‹copies› input into output).

As will be seen later in Section 5.16, I experimented with both automatic and manual loop unrolling to support ILP across multiple work items. The Block DASk also supports experiments on TLP by the initial choices for the CTA layout, For Example: *BlockSize* =

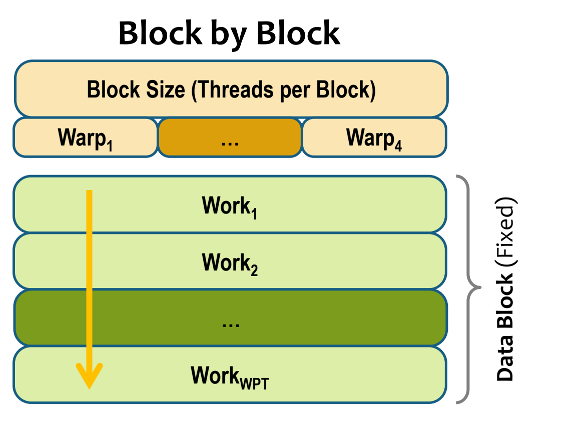
‹128,1,1› and *GridSize* = ‹224,1,1›. To make the Block DASk more efficient, I pick a fixed thread block size (TBS), and a fixed amount of work per-thread (*nWork*), typically in the range [1..4]. From this, the fixed data block size (DBS) can be computed as *DBS*=*nWork*·*TBS*. Once, the data block size is known, the number of data blocks (*m*) needed to cover the entire data set can be computed as $m = \lceil n/DBS \rceil$. A one-to-one mapping between thread and data blocks makes it relatively easy to compute where the corresponding data block starts as *blockOff* = (*bid*·*DBS*). For a 1D grid, the block ID (bid) can be computed as *bid* = *blockIdx.x* and for a 2D grid, *bid* = (*blockIdx.y*·*gridDim.x*) + *blockIdx.x*.

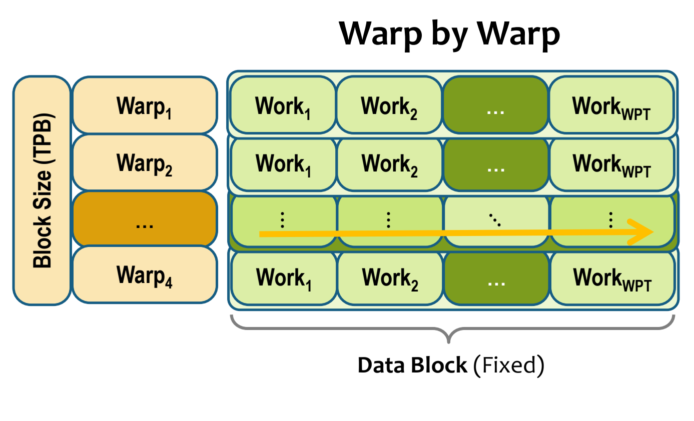
Of course to get higher performance with the Block DASk, I used two main ideas.

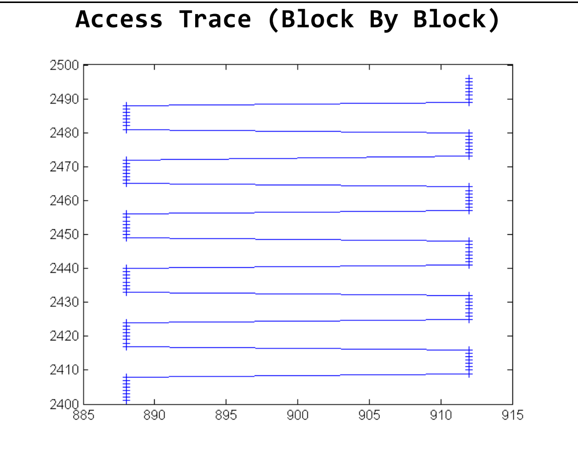
- Coalescence within a data block: All DASks mainly deal with efficient mapping of thread blocks onto data blocks. This corresponds to the first level of the CTA hierarchy (grid of thread blocks). However, an efficient mapping for the second level of the CTA hierarchy (threads within a thread block) is also needed. I introduce two BASks – Block by Block, and Warp by Warp that support coalescence for high throughput.
- Amortizing costs across multiple work items: To increase performance, decrease the amount of instructions. One common technique to reduce instructions is to amortize shared costs across multiple work items. I amortize in two different ways in the Block DASk – amortized range checking and amortized pointer indexing.

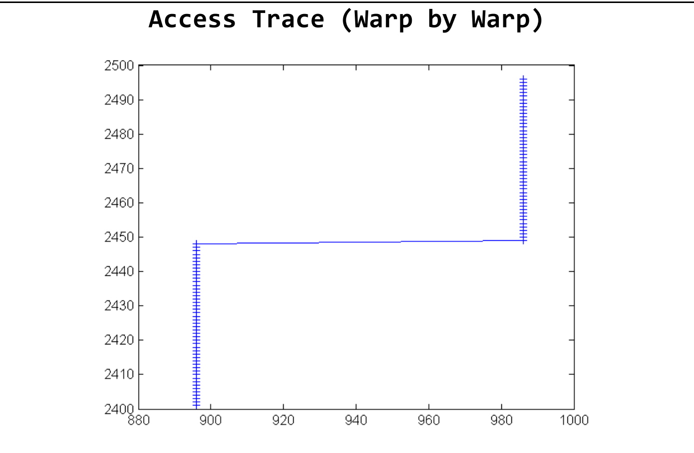
5.1.1 Block Access Skeletons (*Block by Block* vs. *Warp by Warp*):

My Block DASk (all my DASks) provide a framework for the GPU programmer that maps thread blocks onto data blocks, the first level of the CTA hierarchy. To map threads warps onto data warps within each data block, the second level of the CTA, I introduce two *Block Access Skeletons* (BASks) – Block by Block and Warp by Warp. The *Block by Block* BASk is straightforward, in fact, it is the way most GPU programmers immediately think of writing code that supports coalescence, assigning one thread per data element sequentially and then striding (stride = *TBS*) to the next row of data within the data block as needed. The *Warp by Warp* BASk, which also supports coalescence, assigns each thread warp its own fixed size subchunk of work within the data block and moves each thread warp to its starting offset, and then strides (stride = *WarpSize*) through the subchunk of work warp by warp.









```
// Map (block within grid)
...
blockOffset = bid*DBS;
...
// Map (thread within block)
tid = threadIdx.x;
currIdx = blockOffset + tid;

// Do work (Access data)
// Unroll for better performance
for (i = 0; i<nWork; ++i)
   out[currIdx] = in[currIdx];

   // Move to next block of data
   currIdx += ThreadBlockSize;
end for
```

```
// Map (block within grid)
...
// Map (thread within warp within block)
nWarps = ThreadBlockSize/WarpSize;
WPW = nWork*WarpSize;   // Work per Warp
warpRow = tid / WarpSize;
warpCol = tid % WarpSize;
localOffset = (warpRow*WPW)+warpCol;
currIdx = blockOffset + localOffset;

// Do Work (Access data)
// Unroll for better performance
for (i = 0; i<nWork; ++i)
   out[currIdx] = in[currIdx];

   // Move to next warp of data
   currIdx += WarpSize;
end for
```

Figure 5.3: *Block Access Skeletons* (BASKs): The left and right columns represent the *Block by Block* and *Warp by Warp* BASKs respectively. The top, middle, and bottom panels represent conceptual layouts, data access traces and corresponding code snippets for both BASKs respectively.

In Figure 5.3, I highlight the main differences between the Block by Block and Warp by Warp BASks. For a fixed size thread block of size TBS, each thread is responsible for processing (copying) exactly *nWork* work items, which fully covers the DBS data elements in the current data block.

The Block by Block memory access pattern strides through the data block cooperatively by all threads in the thread block. Conceptually, the Block by Block BASk can be thought of as a vertical access pattern on a 2D block, where the number of rows is equal to *nWork* and the length of each data row is equal to TBS. Each thread within the thread block is assigned a single column and then strides to the next assigned row of data elements (*stride* = *TBS*) in a block-by-block manner.

The Warp by Warp BASk assigns a fixed-size data sub-chunk of work to each individual thread warp, with the work per warp (WPW) computed as $WPW = nWork \cdot WarpSize$. Conceptually, the Warp by Warp BASk can be thought of as a horizontal access pattern on a 2D block layout, where the number of data rows is equal to the number of warps per thread block computed as $nRows = TBS/WarpSize$, and the length of each data row is equal to *WPW*. Each thread warp is assigned one data row and strides through its row one data warp at a time ($stride = WarpSize = 32$).

When using the Warp by Warp BASk, the GPU programmer needs to know the warp that each thread belongs to. This information is not directly available but can be derived easily from the thread's ID (tid). For 1D fixed-size thread blocks, I compute the *warpRow*, *warpCol* from the thread ID as ($warpRow = tid/32$, $warpCol = tid\%32$) respectively, where 32 represents the number of threads per warp (*WarpSize*). To avoid magic numbers in my GPU kernels, I convert the *WarpSize* into a fixed-size C++ template parameter as well. Since *WarpSize* is a power of two, the more expensive modulus and division operations can be replaced with simpler bit shift and mask operations instead as ($warpRow = tid >> logWarpSize$, $warpCol = tid \& WarpMask$), where *logWarpSize* and *WarpMask* are computed as $logWarpSize = log_2(WarpSize) = 5$ and

WarpMask = *WarpSize*-1 = 31 respectively. An even simpler alternative is to define the fixed-size thread block layout as 2D instead of 1D (meaning, *Block* = ‹32, *nWarps*, 1› where *nWarps* is the number of warps in the thread block). For example, an 1D thread block of 128 threads could be represented in 2D as ‹32,4,1›. This allows the warp row and column to be read directly from the `threadIdx` CTA parameter as *warpCol* = `threadIdx.x` and *warpRow* = `threadIdx.y`.

Figure 5.3 (middle row) shows memory traces that reveal differences in the access patterns of these two BASks. The specific experiment generating these traces used two warps (*TBS*=64) and 6 work items per thread (*nWork* = 6) per data block. Each warp is represented by eight small plus signs per warp.

The Block by Block BASk (middle row, left panel), ping-pongs as the two warps stride through their respective six data elements; the Warp by Warp BASk (middle row, right panel) has a more sequential trace layout for each warp's assigned sub-chunk of work. Though both BASks work well, most GPU programmers gravitate towards the Block by Block BASk as it is simpler, easier to code and easier to reason about. I personally prefer the Warp by Warp BASk in my code for three reasons:

- The data access pattern is slightly more localized, which can result in a modest increase in I/O throughput (1-2% faster) over the Block by Block BASK despite the extra setup and indexing overhead.
- Each individual warp can proceed on its assigned subchunk of work independent of any other warp in the thread block. With the Block by Block BASk often each warp must wait for all warps to complete transferring data before useful processing work can proceed.
- Barrier Synchronization is often needed for correct communication and coordination across the threads in a thread block. The Warp by Warp pattern BASk typically requires fewer barriers then the Block by Block BASk for correct behavior. This is a

direct result of each warp being able to start working on its assigned subchunk of work independently of other warps. Only when intermediate results need to be shared across the entire thread block does a barrier need to be inserted into the code.

5.1.2 Amortized Range Checking:

Recall that the simple Copy kernel (from Chapter 4) must range-check all data accesses because the last data block (for 1D grids) and last data row (for 2D grids) may only be partially full of data. The Block DASk either needs to range check or pad the data to block boundaries; I have experimented with both.

```
// CTA Mapping
...
blockStart = bid*DBS;
...
// Range check (entire data block)
inRange  = ((blockStart+DBS) < n);
outRange = (blockStart >= n);
if (inRange)
  // In range, process entire block (no further range checks needed)
  ...
else if (outRange)
  // Out of range, Exit kernel (nothing to do)
  ...
else
  // Overlaps, process block (with careful range checks on each access)
  ...
end if
```

Figure 5.4: Amortized range checking for Block DASk.

Because the Block DASk can process several consecutive work items per thread it can amortize range checks across all (*nWork*) work items per thread within each data block. As shown in figure 5.4, there are three cases for range checking consecutive work items (in-range, out-of-range, and overlap):

- *In-range*: The entire data block is inside the valid data range [0,*n*). To decide if the entire data block is in-range, I perform the following test, (*blockStart*+DBS) < *n*. If in-range, all *nWork* data elements per thread can be transferred with no further range checks.
- *Out-of-range*: The entire data block is outside of the valid data range [0,*n*). To decide if the entire data block is out-of-range, I perform the following test, *blockStart* ≥ *n*. If out-of-range, the code can safely exit the kernel without doing any more unnecessary work.
- *Overlap*: One data block may partially overlap the range, requiring more fine-grained per work-item range checks. If a specific data block is neither in-range nor out-of-range, it must partially overlap one of the range boundaries. Consequently, each of the *nWork* data transfers must be carefully range-checked before being allowed to proceed. This overlap case often results in branch divergence but only for this single overlapping data block. Given enough other data blocks, the higher cost of range-checking this single overlapped block gets amortized away.

Padded Access: If the input and output arrays are under the programmers control then data can be padded (filled) to data block or data row boundary with extra sentinel (meaning "do not care") values. By construction, the padded input size is now evenly divisible by the fixed-size data block (or fixed size data row), so no range checks are needed inside the GPU kernel code. I call this approach *padded access*. Amortized range checking reduces range checks, padded access completely eliminates the need for them. There are two main issues with padded access.

First, padded access has costs. The GPU programmer still needs to compute the padded array sizes, allocate padded memory arrays, move data from the original input arrays into the padded input arrays, process the padded array, move the padded outputs back into the original output array (while ignoring outputs outside the original valid range), and then de-allocate any padded arrays. These overhead costs may outweigh any savings obtained by completely eliminating range checking from the GPU kernel(s).

Second, padded access negatively impacts modularity. It requires tight coupling between the GPU kernel and its corresponding CPU host function. For correct behavior, both the GPU kernel and CPU host function must compute data block sizes, and data block (or data row) boundaries based on the original input size (*n*) and CTA layout parameters and then pad the input/output arrays up to these boundaries. This tight coupling between the CPU function and GPU kernel means that a padded access GPU kernel cannot safely be reused in some other context without also porting over the corresponding CPU host code as well.

As a result of the extra overhead costs and tight coupling, I do not recommend using padded access to eliminate range checking.

5.1.3 Amortized Pointer Indexing:

For a minor performance boost and better ILP, I also amortize the cost of pointer setup and indexing across multiple work items within each thread. For the simple Copy kernel (in Chapter 4), I used *absolute* addressing, where a single index per work item is computed and is then used to directly index into the source (*S*) or destination (*D*) arrays. Starting with the BLock DASk, I switch to *base plus offset* addressing, where a single base index is computed, a single base pointer is computed, and then a constant offset per work item is included as each thread accesses the source and destination base pointers for each work item.

As shown in Figure 5.5, amortizing pointer setup and indexing across multiple work items using *base+offset* addressing requires fewer instructions overall. In addition, there are far fewer read-after-write (RAW) dependencies between instructions in the amortized code than with the non-amortized code, and fewer dependencies also results in better ILP performance.

Non-Amortized Pointers

Uses *Absolute* Addressing

```
// Compute 4 indices
w1 = dataOff + (0u*WarpSize);
w2 = dataOff + (1u*WarpSize);
w3 = dataOff + (2u*WarpSize);
w4 = dataOff + (3u*WarpSize);
...

// Access 'source' pointer with 4 indices
v1 = S[w1];
v2 = S[w2];
v3 = S[w3];
v4 = S[w4];
...
```

Amortized Pointers

Uses *Base* + *Offset* Addressing

```
// Compute starting pointer
const valT * in  = &(S[dataOff]);
...
// Access pointer with 4 constant offsets
v1 = in[(0u*WarpSize)];
v2 = in[(1u*WarpSize)];
v3 = in[(2u*WarpSize)];
v4 = in[(3u*WarpSize)];
...
```

Generated PTX code

```
mul.wide.u32   %w1_64, w1, 4;
add.s64        %in1, %dataOff, %w1_64;
ld.global.u32  %v1, [%in1];

mul.wide.u32   %w2_64, w2, 4;
add.s64        %in2, %dataOff, %w2_64;
ld.global.u32  %v2, [%in2];

mul.wide.u32   %w3_64, w3, 4;
add.s64        %in3, %dataOff, %w3_64;
ld.global.u32  %v3, [%in3];

mul.wide.u32   %w4_64, w4, 4;
add.s64        %in4, %dataOff, %w4_64;
ld.global.u32  %v4, [%in4];
...
```

Generated PTX code

```
// Compute starting pointer
mul.wide.u32   %dOff_64, dataOff, 4;
add.s64        %in, %S, %dOff_64;
...

// Access pointer with 4 constant offsets
ld.global.u32  %v1, [%in+0];
ld.global.u32  %v2, [%in+128];
ld.global.u32  %v3, [%in+256];
ld.global.u32  %v4, [%in+384];
...
```

Figure 5.5 Amortized pointer indexing. The left panel doesn't amortize pointer indexing and uses *absolute* addressing. The right panel amortizes pointer indexing across multiple work items and uses *base+offset* addressing. The equivalent generated .PTX for each is included in the lower panels.

In some of my other case studies on earlier versions of the CUDA platform, rewriting code to amortize pointer indexing made a noticeable difference in performance. However, the actual performance improvements I saw for the Copy primitive using CUDA 5.5 were very modest, for $n=2^{24}$ on a GTX Titan, The non-amortized absolute addressing `Copy_Block` kernel runs at ~234 GB/s, and the amortized base+offset addressing kernel runs at ~235 GB/s. This tells me that the CUDA compiler has matured in the past couple of versions.

5.1.4 Block DASk Code

The *Block* DASk enhances the simple copy I/O kernel with C++ template parameters, multiple work items per thread, amortized range checking, amortized pointer indexing, and manual loop unrolling, as shown in Figure 5.6.

```
template < valT, logWarpSize, BlockSize, nWork >
__global__ void Copy_Block( D, S, start, stop ) {
  // Compiler-Time Variables (become constants at run-time)
  const U32 WarpSize = 1u << logWarpSize; // Threads per Warp (32)
  const U32 WarpMask = WarpSize - 1u;     // Warp Mask (31 = 32-1)
  const U32 DBS      = nWork*BlockSize;     // Data per Block (512 = 4*128)
  const U32 DWS      = nWork*WarpSize;      // Data per Warp  (128 = 4*32)
  const U32 DB_LAST  = DBS-1u;            // Last item in data block (511)
  const U32 DW_LAST  = DWS-1u;            // Last item in data warp  (127)

  // Map CTA parameters onto data offsets
  U32 bid = (blockIdx.y * gridDim.x) + blockIdx.x;  // Block ID (bid)
  U32 tid = threadIdx.x;            // Thread ID (tid)
  U32 warpRow = tid >> logWarpSize; // tid / WarpSize
  U32 warpCol = tid & WarpMask;     // tid % WarpSize
  U32 warpOff  = (warpRow * DWS) + warpCol; // starting warp  offset
  U32 blockOff = (bid * DBS) + start;       // starting block offset
  U32 dataOff  = blockOff + warpOff;        // starting data offset
  U32 blockStart = blockOff;
  U32 blockStop  = blockOff + DB_LAST;

  // Get start pointers (input & output)
  const valT * in  = &(S[dataOff]);
        valT * out = &(D[dataOff]);

  // *data block range checks* against [start, stop]
  bool inRange  = (blockStop <= stop);
  bool outRange = (stop < blockStart);
  if (inRange) {
    // #1) In Range, process entire block (with *NO* range checking)
    valT v1, v2, v3, v4;

    // Load values
    if (nWork >=  1u) { v1 = in[( 0u*WarpSize)]; }
    if (nWork >=  2u) { v2 = in[( 1u*WarpSize)]; }
    if (nWork >=  3u) { v3 = in[( 2u*WarpSize)]; }
    if (nWork >=  4u) { v4 = in[( 3u*WarpSize)]; }

    // Store values
    if (nWork >=  1u) { out[( 0u*WarpSize)] = v1; }
    if (nWork >=  2u) { out[( 1u*WarpSize)] = v2; }
    if (nWork >=  3u) { out[( 2u*WarpSize)] = v3; }
    if (nWork >=  4u) { out[( 3u*WarpSize)] = v4; }

  } else if (outRange) {
    // #2) Out of Range, Exit Kernel (nothing to do)
  }
  else { // Overlaps case
    // #3) Overlaps Case, process block (with careful range checking)
    U32  D1, D2, D3, D4;
    bool T1, T2, T3, T4;
    valT v1, v2, v3, v4;

    // Get data offsets
    if (nWork >= 1u) { D1 = (0u*WarpSize) + dataOff; }
    if (nWork >= 2u) { D2 = (1u*WarpSize) + dataOff; }
    if (nWork >= 3u) { D3 = (2u*WarpSize) + dataOff; }
    if (nWork >= 4u) { D4 = (3u*WarpSize) + dataOff; }

    // Range Check (each data offset)
    if (nWork >= 1u) { T1 = (D1 <= stop; }
    if (nWork >= 2u) { T2 = (D2 <= stop; }
    if (nWork >= 3u) { T3 = (D3 <= stop; }
    if (nWork >= 4u) { T4 = (D4 <= stop; }

    // Load values (with range checking)
    if (nWork >=  1u) { if (T1) { v1 = in[(0u*WarpSize)]; } }
    if (nWork >=  2u) { if (T2) { v2 = in[(1u*WarpSize)]; } }
    if (nWork >=  3u) { if (T3) { v3 = in[(2u*WarpSize)]; } }
    if (nWork >=  4u) { if (T4) { v4 = in[(3u*WarpSize)]; } }

    // Store values (with range checking)
    if (nWork >=  1u) { if (T1) { out[(0u*WarpSize)] = v1; } }
    if (nWork >=  2u) { if (T2) { out[(1u*WarpSize)] = v2; } }
    if (nWork >=  3u) { if (T3) { out[(2u*WarpSize)] = v3; } }
    if (nWork >=  4u) { if (T4) { out[(3u*WarpSize)] = v4; } }

  } // end *data block range checks*
} // end Copy_Block
```

Figure 5.6: My *Block* DASk for the Copy primitive. The code outside the three shaded boxes makes up the scaffolding for data access. The code inside the shaded boxes (#1, #3) would be replaced by equivalent code to support the GPU programmers own algorithm.

There are five things that I note from reviewing my Block DASk code.

First, the kernel is a combination of a framework skeleton and the GPU programmer's body to perform the desired Copy operation. The code (outside of the shaded boxes) provides the DASk framework that supports efficient data access patterns on the GPU. The body representing the GPU programmer's code is contained (in the shaded boxes in Figure 5.6 and throughout my code figures in this thesis). The programmer can easily change the code in the shaded boxes to support different algorithms such as (Fill, Gather, Scatter).

Second, even though this DASk is more complicated than the simple copy kernel, it is not overly complex. Some of the additional lines of code support the amortized range checking pattern on each data block. Others use manual loop unrolling to support multiple work items per thread.

Third, amortized range checking increases the number of shaded boxes for the GPU programmer to fill in since there are three cases (in-range, out-of-range, and overlap), two of which require code: in-range copies *nWork* items without range checking, and overlap copies *nWork* items with careful range checking.

Fourth, this example kernel code supports only 1-4 work items per thread. My actual kernel is even longer because, although I use this same DASk, I increase manual loop unrolling by replicating code within the shaded box to support up to 16 work items per thread in batches of four work items to control register pressure.

Fifth, the `if` statements based on the constant template parameter *nWork* are evaluated at compile time, so only one assignment `v1=in[…];` is compiled.

5.1.5 Improving Copy Performance using ILP and TLP

As we saw in chapter 4, my simple Copy kernel had poor performance. In this section, I seek to improve Copy performance by hiding stalls in the GPU hardware, such as waiting on long I/O operations (transfers between registers and global memory takes between 400-800 cycles) or waiting on the previous instruction's output (RAW dependency takes up to 8-11 cycles). I hide stalls using a combination of ILP and TLP to keep the SPs on each SM as busy as possible.

Increasing ILP means increasing the work done per-thread. To test ILP, I wrote code that tested both automatic and manual loop unrolling. As will be seen shortly, manual loop unrolling, also known as software pipelining, gives better performance.

Increasing TLP means increasing the threads-per-block. This increases opportunities for the warp scheduler to hide stalls using independent instructions from other concurrent warps.

Since my simple Copy kernel did not take advantage of either ILP or TLP, it achieved only 25% of the maximum peak throughput and was about 2.7x slower than the build-in CUDA library function `cudaMemcopy`. Contrast this with Figure 5.1 where each of my three Copy DASks achieves throughput performance comparable to `cudaMemcopy`.

Increasing ILP: In this section, the programmer will learn how to use manual loop unrolling in their own kernels, I demonstrate this using my Block DASk for the Copy primitive. For this Block Copy kernel, Manual loop unrolling results in up to 17% better performance than my simple Copy kernel from Chapter 4.

Loop Unrolling: Loops are the classic programming technique to handle multiple work items in CPU serial code. *Loop unrolling*[3] hides stalls caused by dependences between instructions and between loop iterations by rewriting the loop to process more independent data elements inside of each loop iteration. Loop unrolling amortizes loop overhead across k elements and also amortizes the cost of indexing and pointer computations. This technique can either be implemented automatically via a compiler or manually by the programmer. As each independent work item may require registers to track execution state, I suggest unrolling data in small batches of 2-8 work items to avoid exceeding the number of registers and spilling into local memory.

Processing more work items per thread is the GPU kernel equivalent of loop unrolling. Loop unrolling is just one of a family of optimization techniques (Wadleigh and Crawford, 2000) used to improve the performance of loops. These include loop fission, loop fusion, loop interchange, loop invariant code motion, loop unrolling, loop reversal, and loop unswitching. Almost the entire family of serial loop optimizations can be repurposed and rebranded as GPU kernel optimizations for parallel programming. Recall in Chapter 4.1 that we related serial iteration to data parallelism (meaning that the innermost body of statements within some nested loop structure where the looping of n iterations across n data elements is replaced by hardware scheduling of n parallel threads onto n data elements.)

[3] As described in the book Software Optimization for High Performance Computing (Wadleigh and Crawford, 2000).

Automatic Loop Unrolling: A specific example of automatic loop unrolling is given in Figure 5.7. This code would be used in the GPU Programmers shaded boxes in Figure 5.6 to replace the current software pipelined copy code shown. The CUDA compiler supports a `#pragma unroll` directive that takes an optional batch size parameter to specify the number of iterations to unroll the loop (see the line occurring before source line #7 for an example).

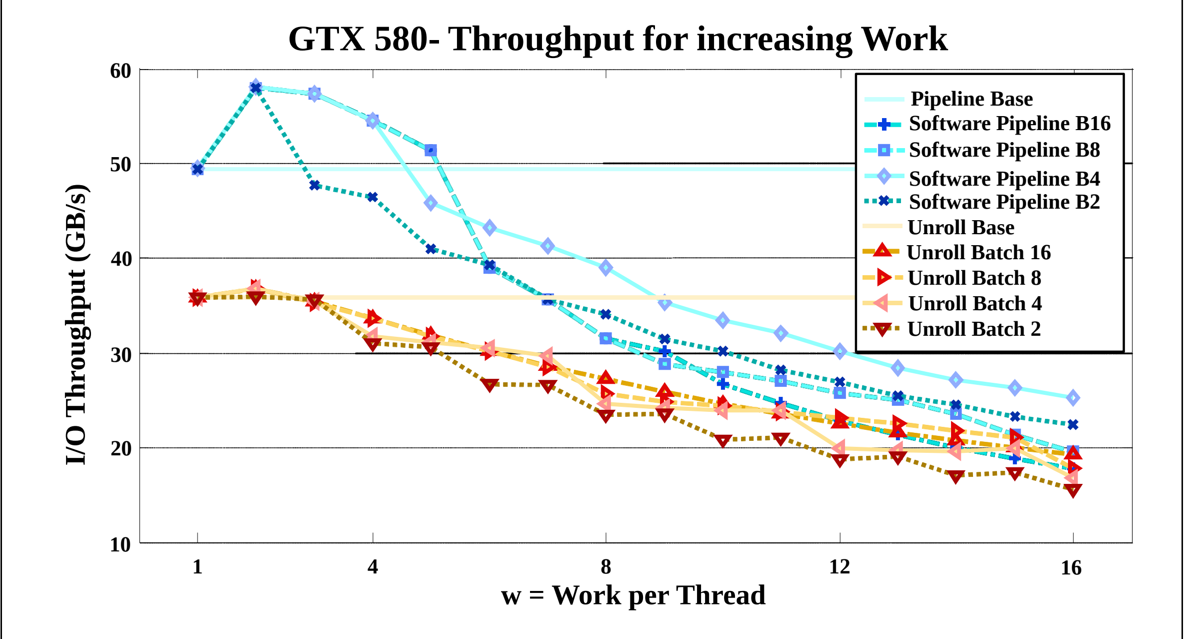
```
     // Copy assigned work items (nWork)
     #pragma unroll 4
 1:  for (i=0; i<nWork; ++i)
 2:    wOff = (i*TBS)+dOff; // Work Item Offset
       // Range check [start, stop]
 3:    inRange = (start ≤ wOff) & (wOff ≤ stop);
 4:    if (inRange)
 5:      D[wOff] = S[wOff]; // Copy Work Item from input to output
 6:    end if
 7:  end for
```

Figure 5.7: *Automatic Loop unrolling* example: The `#pragma unroll 4` directive (in lighter grey) around a looping structure requests CUDA to automatically unroll the wrapped code (*k*=4*)* times

For my loop unrolling example, I rewrote my Block DASk do the loop unrolling using the `#pragma unroll` directive. All necessary changes to support loop unrolling replace the code found in the shaded boxes in Figure 5.6. The shaded box represents the user's portion of this simple DASk, in other words, the code that another GPU programmer would change to support a different algorithm.

After testing the automatic loop unrolled copy, I found that this approach resulted in only a modest improvement in throughput as compared to a baseline of one work-item (see Figure 5.8). The change was barely noticeable on the GTX 580 (+0.25% in the best case) and not much better on the GTX Titan (+8.8% in the best case). As can be seen from the graph, only [2-3] work items per thread results in a minor throughput performance boost, which drops off gradually after four work items per thread. Also note that the automatic unrolled kernel's throughput (~61 GB/s) is actually well short of the original simple Copy kernel's throughput (~81 GB/S) on the

GTX Titan. This decreased throughput is caused by a poor memory access pattern that I will explain later in this section.



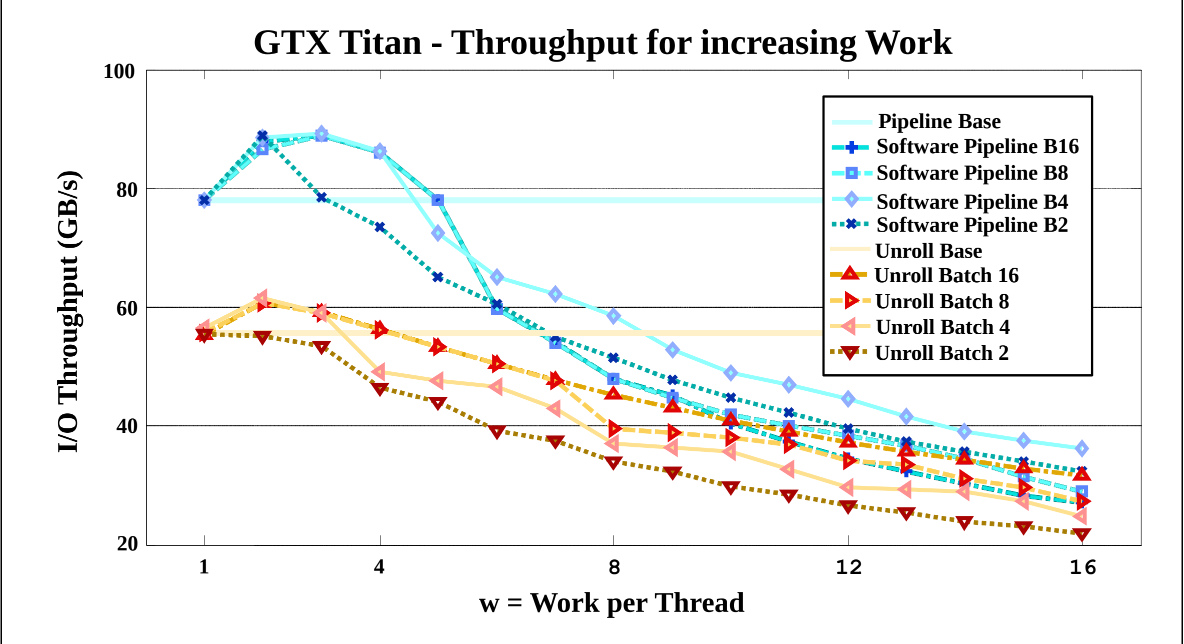


Figure 5.8: *Automatic Loop unrolling* vs. *Manual Loop Unrolling* test throughput: Given a fixed input size $n=2^{24}$, fixed grid row size = 224, fixed block size = 32, I/O throughput (in GB/s) is shown on the *y*-axis and the amount of work per thread, *nWork* = [1-16], is shown on the *x*-axis. The upper & lower panels show throughput results on the GTX 580 (Fermi) and GTX Titan (Kepler) GPUs respectively. The cyan/blue lines represent manual loop unrolling and the tan/red lines represent automatic loop unrolling Batching for both automatic and manual loop unrolling was tested in batches of [2,4,8,16] respectively.

Manual Loop Unrolling: Next, I loop unrolled my code by hand, I interleaved *k* instructions from *k* work items to decrease RAW dependencies. The main idea here is that, while a particular instruction for the *i*th work item may stall, similar instructions from the other *k*-1 work items can be scheduled as replacements to hide the stall and keep each processing core busy doing useful work. This form of manual loop unrolling is also known as *software pipelining*[4]. More independent work items increases register pressure which in term may limit occupancy, so I experimented with manual unrolling (on up to 16 work items) in batches of [2, 4, 8, or 16]. Figure 5.9, shows Copy code that range-checks multiple work items using manual loop unrolling. This is the type of code that a GPU programmer would insert into the shaded boxes of Figure 5.6 to support the Copy primitive.

[4] As described in the book Software Optimization for High Performance Computing (Wadleigh and Crawford, 2000).

```
// Process work items [1-4]
  // Get work offsets
    if (nWork≥1) { w1 = (0*TBS)+dOff; }
    if (nWork≥2) { w2 = (1*TBS)+dOff; }
    if (nWork≥3) { w3 = (2*TBS)+dOff; }
    if (nWork≥4) { w4 = (3*TBS)+dOff; }
  // Range check [start, stop]
    if (nWork≥1) { t1 = (start ≤ w1) & (w1 ≤ stop); }
    if (nWork≥2) { t2 = (start ≤ w2) & (w2 ≤ stop); }
    if (nWork≥3) { t3 = (start ≤ w3) & (w3 ≤ stop); }
    if (nWork≥4) { t4 = (start ≤ w4) & (w4 ≤ stop); }
  // Load data
    if (nWork≥1) { if (t1) { v1 = D[w1]; } }
    if (nWork≥2) { if (t2) { v2 = D[w2]; } }
    if (nWork≥3) { if (t3) { v3 = D[w3]; } }
    if (nWork≥4) { if (t4) { v4 = D[w4]; } }
  // Store data
    if (nWork≥1) { if (t1) { S[w1] = v1; } }
    if (nWork≥2) { if (t2) { S[w2] = v2; } }
    if (nWork≥3) { if (t3) { S[w3] = v3; } }
    if (nWork≥4) { if (t4) { S[w4] = v4; } }
```

Figure 5.9: *Manual Loop Unrolling* example, showing how to copy multiple work items using careful range checking. This example assumes at most four work items per thread. Notice how the similar instructions are batched together in groups of 4 in an interleaved manner. The lighter grey `if (nWork ≥ ?) { … }` statement wrappers get elided away at compile time by the CUDA compiler.

The `if (nWork≥*) {…}` wrappers are resolved at compile time. Hand unrolled code is more verbose, harder to read, and harder to understand. In addition, the up to *k*× as many generated instructions may also use *k*× as many registers.

Loop Unrolling Results: Tests on both automatic and loop unrolling are shown in figure 5.8. To stay in the stable upper portion of the s-shaded throughput curves, a fixed input size of n=2^{24} was chosen. To show the impact of multiple work items on performance, *nWork* was increased from 1-16. To show the impact of register pressure, work items were batched into groups of [2,4,8, and 16]. All throughput numbers presented are averages of one hundred runs.

Figure 5.8 shows four surprising things.

First, there is a large difference in starting throughput for the baseline case (*nWork* = 1) for the automatic vs. manual loop unrolling. This difference in starting throughput is the result of the different memory access patterns used by the two different methods. For the automatic loop unrolled kernel, I wrote each C++ copy instruction as `if (inRange) { S[wOff] = D[wOff] }`. This line of code performs two memory accesses: one load from input and one store to output and is then repeated *k* times, in other words, the memory access pattern ping-pongs back and forth between the input and output arrays. However, in the manually unrolled kernel, the code batches the loads and stores separately as two different instruction clusters (*k* inputs followed by *k* outputs). To verify that the performance difference was indeed a result of these two different memory access patterns, I rewrote my manually unrolled kernel as a set of `if (t1) { S[w1] = D[w1]; }` statements. This approach resulted in a large decrease in throughput similar to the automatically unrolled code kernel. So it seems clear that batching up several warps of data accesses (input **or** output) in close proximity to each other results in better system throughput from the memory controllers than interleaving access to different parts of memory (input **and** output). Such a conclusion makes sense since better locality improves L1 and L2 cache usage by each GPU memory controller.

Second, the automatic loop unrolling batch curves separate one work item sooner than I expected. For example, for both automatic and manual loop unrolling I would expect the *nWork* throughput curve for batching in groups of 4, 8, and 16 to be exactly the same up to *nWork* = 4 and then begin to separate from each other at *nWork* = 5 and *nWork* = 9. This expected behavior is observed in the manual loop unrolling throughput curves. However, the separation for loop unrolling occurs sooner at *nWork* = 4 and *nWork* = 8 for the automatic loop unrolling throughput curves. I speculate that there could be an "off by one" bug in the CUDA compiler in this case.

Third, the performance curves drop off throughput more rapidly than I was expecting. I was expecting that lower performance due to lower occupancy caused by increased register usage wouldn't show up until 8+ work items per thread. The register pressure effect can be seen for the *Batch* = 16 manually unrolled curve which has the worst performance for large work loads (*nWork* > 8), whereas the *Batch* = 4 and *Batch* = 8 manually unrolled curves have better performance for (*nWork* > 8). But, throughput drops off quickly after only 4 work items per thread with the largest drop occurring between *nWork* = 5 and 6. I suspect that there is also some memory controller queue length or caching issue coming into play here. Surprisingly, for the automatic unrolled throughput curves, The *Batch* = 16 curves have some of the best performance for *nWork* = 16. For instance, a single warp loading 4 work items would fetch 4 warp lines (512 bytes) of data. It could be that the combined combination of 14 SM's each with 16 thread blocks of 128 threads (4 warps) requesting 6 work items per warp (5,376 warp line requests in aggregate) is overflowing the internal request queues in the six memory controllers (896 requests per controller on average) for each set of active concurrent thread blocks on each SM).

Fourth, manual unrolling results in better throughput than automatic unrolling. Since the results for loop unrolling and software pipelining have different starting throughputs, I decided to use the *nWork* = 1 case as a baseline for both cases. Comparing the throughput results to their respective baselines reveals that software pipelining improves throughput more than loop unrolling. As the following table shows, the maximal throughput increase when manual unrolling was used was substantially greater for both GPUs than when automatic unrolling was used.

Throughput Increase	Loop Unrolling	Software Pipelining
GTX 580	0.25%	17%
GTX Titan	7%	14%

Table 5.1: Loop Unrolling vs. Software Pipelining Performance

This makes sense since my manually unrolled code batches similar instructions into groups to reduce data dependencies (grouping and interleaving instructions from *k* work items). This organization provides plenty of independent instructions for the static compiler or dynamic warp scheduler to exploit in order to hide stalls. As the CUDA platform continues to mature, loop unrolling should eventually done automatically by the compiler instead of manually by the programmer. For now, it pays off for the GPU programmer to manually unroll and batch their instructions for up to 3-4 work items.

Increasing ILP is not the only way to hide stalls, as we will see next. TLP techniques work even better.

Increasing TLP: Another way to hide stalls uses TLP to recycle instructions from other independent and concurrent warps of execution. Fermi supports up to 48 warps (1,536 threads) per SM and Kepler supports up to 64 warps (2,048 threads) per SMX. In this section, I show how increasing the amount of threads per thread block can increase throughput up to 216 GB/s (2.67× faster than the simple Copy from chapter 4).

One of the main reasons that the simple Copy in Chapter 4 had poor throughput is that the code naively did not take advantage of the massive parallelism via TLP available on GPU architectures. The simple Copy only launched with a CTA layout of only 32 threads (TBS=32) only achieving an occupancy of 16.67% (8/48) and 25% (16/64) on the GTX 580 and GTX Titan respectively. I already discussed Occupancy and constraints in Chapter 3.3., recall that occupancy is a simple ratio between the number of thread warps that actually concurrently run on each SM for a given GPU kernel vs. the theoretical maximum number of warps that could concurrently run on a given architecture.

Varying the number of warps on each SM is actually tricky since the programmer has no direct control over the number of concurrent warps that are actually scheduled on each SM. The GPU programmer can directly specify the number of threads per block as part of the CTA layout. However, the CUDA platform schedules as many concurrent thread blocks as it can while staying within various resource constraints.

Thus a GPU programmer can request a certain number of threads per thread block (TBS) and a certain number of thread blocks per grid (*GridSize*) but the actually occupancy is determined by the SM scheduler based on the maximum warps, maximum blocks, register per thread usage, and shared memory per block usage. Since, my Grid-based Copy kernels do not use any shared memory and is relatively simple code, requiring just a few registers, CUDA will launch as many concurrent thread blocks on each SM as allowed by the maximum warps constraint. For full processor utilization, I want all SMs running with as many concurrent thread blocks as reasonably possible. This means that I should request my grid size (blocks per grid) to be a multiple of (#*SMs* × #*Blocks*), where #*SMs* is the number of SMs (or SMXs) on a specific GPU card and #*Blocks* is the number of concurrent thread blocks that I expect the CUDA platform to schedule onto each SM.

However, all the SMs also share a limited number of memory controllers on each GPU card (6 on the GTX 580 and 4 on the GTX Titan). Therefore, too many I/O requests may overload the memory controllers, cause cache thrashing, and actually slow down overall throughput. So, my experiments vary the number of warps per thread block and also varying the number of concurrent thread blocks per SM in order to find an optimal balance between thread concurrency and memory controller throughput. To stabilize performance in the upper part of the s-shaped throughput curves, a fixed input size of $n=2^{24}$ data elements is used for all of the following experiments.

To make it easier to experiment with varying the number of threads, My Block DASk for the Copy kernel supports the following four C++ template parameters. The `valT` template parameter specifies the underlying data type of the data and gives me generic type support for different data types other than just 32-bit unsigned integers. The `BlockCols` and `BlockRows` parameters specify the fixed number of columns and rows in the thread block, while the `GridCols` parameter specifies the fixed number of columns in the corresponding CTA grid. These three fixed size CTA template parameters allows me to vary the CTA layout without having to rewrite my copy kernel for each new configuration.

Figure 5.10 shows the impact on throughput of my modified Copy kernel as I increase the number of threads per thread block from [32-1024] in multiples of the *WarpSize* (32).

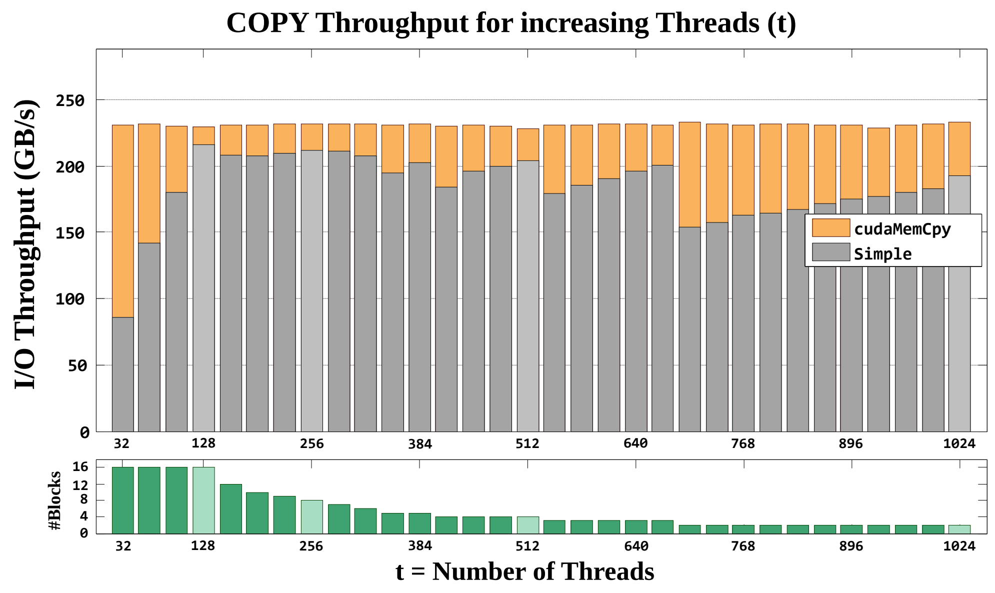


Figure 5.10: Copy throughput for fixed $n=2^{24}$ and increasing *block size* (threads per block) [32-1024]. The top panel shows throughput for a given block size [32-1024] with the orange blocks showing throughput for `cudaMemCopy` and the gray blocks showing throughput for our TLP copy kernel. All measurements are taken as the average of a 100 runs on a GTX Titan. The bottom panel shows the number of concurrent thread blocks per SMX for each matching block size, from [16-2]. Lighter colors are used to show locations where the thread block size divides the maximum threads per SMX exactly.

Throughput from `cudaMemCopy` (orange bars) is also included for comparison, which achieves about 231 GB/s throughput this experiment and remains consistently clustered around this number. However, my modified Copy kernel (gray bars) quickly grows to a peak of 216 GB/s (at 128 threads per block) and then goes into an up and down, saw tooth pattern, where performance drops off a small throughput cliff and gradually climbs back up in increasingly longer runs before dropping off another cliff. Each small cliff corresponds to a drop in the number of concurrent thread blocks that can execute at the same time on each SMX due to the maximum warps per SMX constraint. Only some of the block sizes (32, 64, 128, 256, 512, 1024) actually divide evenly (lighter bars) into the maximum number of threads (2,048) per SMX on a Kepler card such as the GTX Titan. On a Fermi card such as the GTX 580, I would expect a similar pattern with cliffs occurring at 32, 64, 96, 128, 192, 256, 384, 512 since these are the block sizes that evenly divide the 1,536 maximum threads per SM on a Fermi card. That each new performance peak is slightly lower than the previous peak is likely due to locality effects in memory. In other words, as we get more data items per data block each thread warp accesses data warps that are more spread out across each data block in memory. The net result is decreased locality.

In Figure 5.11, I show what happens when I artificially constrain the number of concurrent blocks per SMX in the range [1-16] on the GTX Titan (Kepler architecture). This is done by declaring an unused shared memory array of an appropriate size to restrict the allocated number of concurrent thread blocks per SMX down to the desired number. I had to modify the copy kernel by using an extra Boolean input parameter and an `if` statement using the Boolean input parameter wrapping an unused assignment statement in order to trick the CUDA compiler into keeping the effectively unused shared memory array code around at runtime to constrain concurrency. I also vary the requested columns per grid as a multiple of the number of SMXs times the number of concurrent blocks per SMX (*workLoad* = *nSMs* × *nConBlocks*), so that all the thread blocks are load balanced across the SMXs evenly.

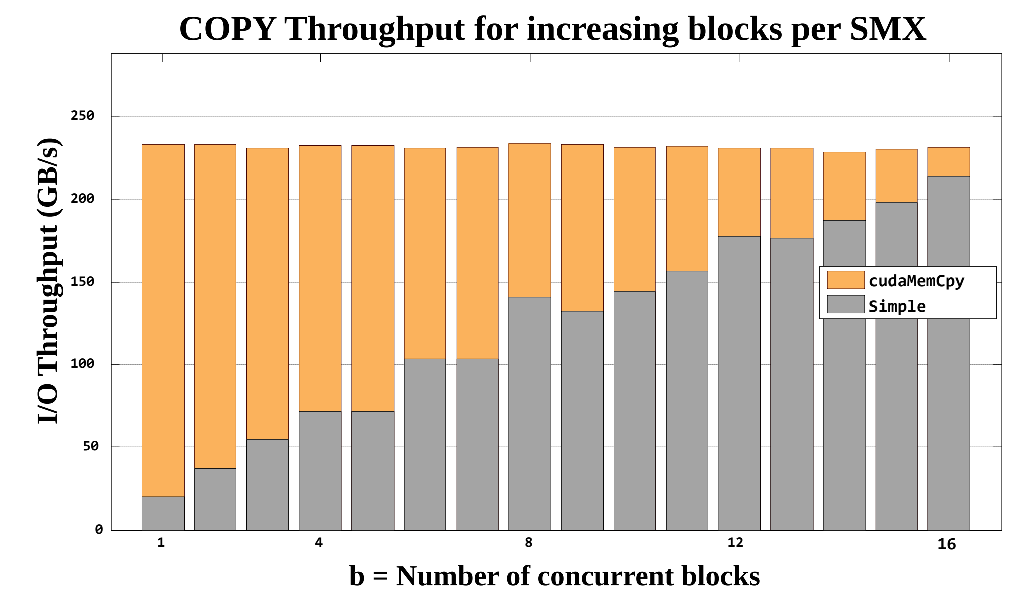


Figure 5.11: Copy throughput for fixed input size n=2^{24} and fixed block size *TBS* = 128, and an increasing number of concurrent blocks per SMX [1-16]. Again orange & gray bars represent I/O throughput for **`cudaMemCopy`** and our TLP copy kernel respectively.

The overall throughput ranges from a low of ~20 GB/s for one concurrent thread block per SMX to a high of ~214 GB/s for sixteen concurrent thread blocks per SMX. Recall that Fermi-based cards only support a maximum of 8 concurrent thread blocks per SM. As easily seen, the overall trend is increasing throughput as the number of concurrent thread blocks increase.

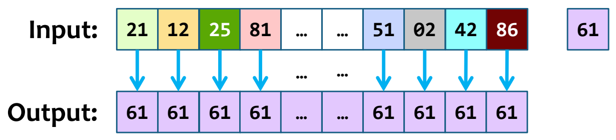
Choosing at least 4 warps per thread block (TBS = 128) results in full occupancy and provides the best throughput. This makes sense, as there lots of thread warps for the SM scheduler to switch between in order to hide stalls.

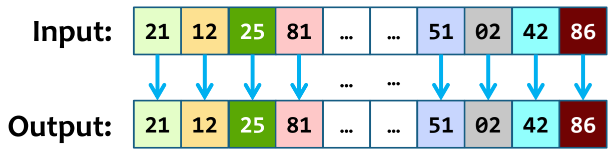
5.1.6 Block DASk Conclusion

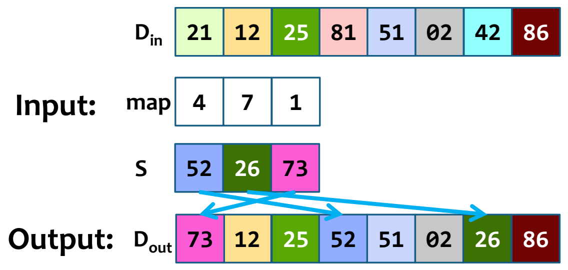
The Block DASk is correct, robust, and has solid I/O throughput. As can be seen from Figure 5.1, the copy implementation using a Block DASk has the best I/O throughput, approaching 81% of peak performance in the best case on the GTX Titan GPU card. The Block DASk can easily be adapted to work with 1D runs, 2D tiles, or 3D blocks, depending on the GPU programmer's problem domain. The Block DASk is extremely suitable for any algorithm that follows the map pattern. However for problems that have a dependency ordering between adjacent neighbors, a more advanced DASk, such as the Row DASk, may be more suited.

5.2 More MAP Primitives

Recall that the Copy primitive is just one example of a map pattern. Table 5.2 shows how to adapt my DASks for other primitives such as Fill, Gather, and Scatter by having the GPU programmer simply change the "body" (= shaded boxes) in my Copy DASks. The code shown is obviously a short hand for performing a single transform to convert a single input into a single output. The programmer will have to write code using loop unrolling or software pipelining to support multiple work items per thread.

`Fill_Block( D, n, toFill )`
DASk Changes:
Eliminate unused source array **S** from Copy_Grid DASk.
User Code Change:
`D[w1] = toFill;`
`...`
	 Input:
	21
	12
	25
	81
	…
	…
	51
	02
	42
	86
	61
	Output:
	61

`Copy_Block( S, D, n )`
DASk Changes: Use as is
User Code Change:
`D[w1] = S[w1];`
`...`
	 Input:
	21
	12
	25
	81
	…
	…
	51
	02
	42
	86
	Output:

`Scatter_Block( S, D, n, map )`
DASk Changes:
Add new **`map`** array parameter.
User Code Change:
`D[map[w1]] = S[w1];`
`...`
	 D_{in}
	21
	12
	25
	81
	51
	02
	42
	86
	Input:
	map
	4
	7
	1
	S
	52
	26
	73
	Output:
	D_{out}
	73
	12
	25
	52
	51
	02
	26
	86

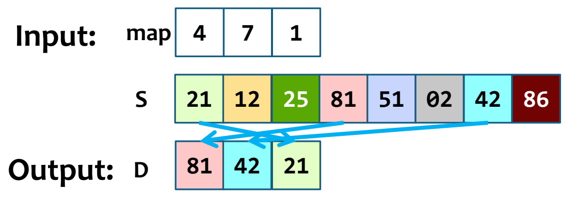
`Gather_Block( S, D, n, map )`
DASk Changes:
Add new **`map`** array parameter.
User Code:
`D[w1] = S[map[w1]];`
`...`
	 Input:
	map
	4
	7
	1
	S
	21
	12
	25
	81
	51
	02
	42
	86
	Output:
	D
	81
	42
	21

Table 5.2: Modifications to my Block DASk to support *Fill*, *Copy*, *Scatter*, and *Gather* primitives

Fill: fills a destination array D of *n* elements with a single data value.

Copy: copies a source array *S* of *n* elements into a destination array *D;* no overlap allowed.

Scatter: scatters elements from a small array into a large array using an index map.

Gather: gathers elements from a large array into a small array using an index map.

All these primitives can use the `Copy_Block` DASk (see Figure 5.6) as a starting framework. I have only made minor changes to each primitive's kernel to make them perform their different functions. The changes to the DASk code are mainly to support removing or adding additional kernel parameters. The user changes are mainly to alter the underlying map transform operators for each primitive. For instance, the scatter and gather operations both require double indexing through an additional `map` look-up parameter, which is added to the kernels list of parameters.

The throughput performance for all map primitives should be proportional to $O(n/p)$, where *n* is the input data set size and *p* is the number of threads (processors). The Fill primitive should be the fastest of the four because it has the best locality (meaning, it only uses a single output array and thus there is no ping-ponging between input and output). Furthermore, since this primitive can be written to use only a single coalesced I/O per data warp, parallel throughput performance should be proportional to $O(\left\lceil \frac{n}{32p} \right\rceil)$. The second fastest primitive should be the Copy primitive, whose locality ping-pongs back and forth between the two arrays (input and output). Since this primitive can be written to use only two coalesced I/Os per data warp, parallel throughput performance should be proportional to $O(\left\lceil \frac{2n}{32p} \right\rceil)$. The Scatter and Gather primitives should be the slowest primitives. They switch locality between three arrays (input, map, output). They can be written to perform two of the three I/Os (scatter → input & map; gather → map & output) in a coalesced manner on each data warp. However, the third I/O (scatter → output;

gather → input) disperses memory accesses across unpredictable memory locations for each individual thread and thus the entire data warps memory accesses are non-coalesced. For both Scatter and Gather, worst-case parallel throughput performance should be proportional to $O(\left\lceil\frac{2n}{32p}\right\rceil + \left\lceil\frac{n}{p}\right\rceil)$, or equivalently to $O(\left\lceil\frac{34n}{32p}\right\rceil)$. The main reason the Fill and Copy map primitives can approach peak throughput on the GPUs is because they fully support coalescence. Whereas, because one I/O out of three in the Scatter and Gather primitives do not support coalescence, these primitives can run much slower (up to 17× slower than Copy).

5.3 *Column* DASk

The Block DASk is only one of many possible ways to coordinate thousands of threads to access data in global memory. The layout used for the Block DASk is fundamentally one dimensional. In this section, I present a fundamentally two dimensional data access pattern (of rows and columns), which I call the *Column* DASk. For the Column DASk, the input data (*n*) is divided into *m* fixed-size data blocks (*DBS*) with the data blocks being laid out on a 2D grid, with the total number of blocks (*m*) needed to cover data computed as *m* = $\lceil n/DBS \rceil$. The number of columns is chosen ahead of time by the GPU programmer as a static fixed size constant (*C*= *grid.width*). The number of rows (*R* = data blocks per column) is allowed to vary with *n* in order to fully over all input data and can be computed as *R* = $\lceil m/C \rceil$. The last row may only be partially full requiring careful range checking to avoid out-of-range data accesses. However, the rest of the full rows require no range checking. The Column DASk maps one thread block onto each data column, with each thread block being responsible for processing all data blocks along its assigned column.

Figure 5.12 shows how the Column DASk for mapping blocks onto a grid is conceptually setup and is quite similar to the Block by Block BASk for mapping thread warps onto a data block (see Section 5.1.2 for a more detailed description of the Block by Block BASk).

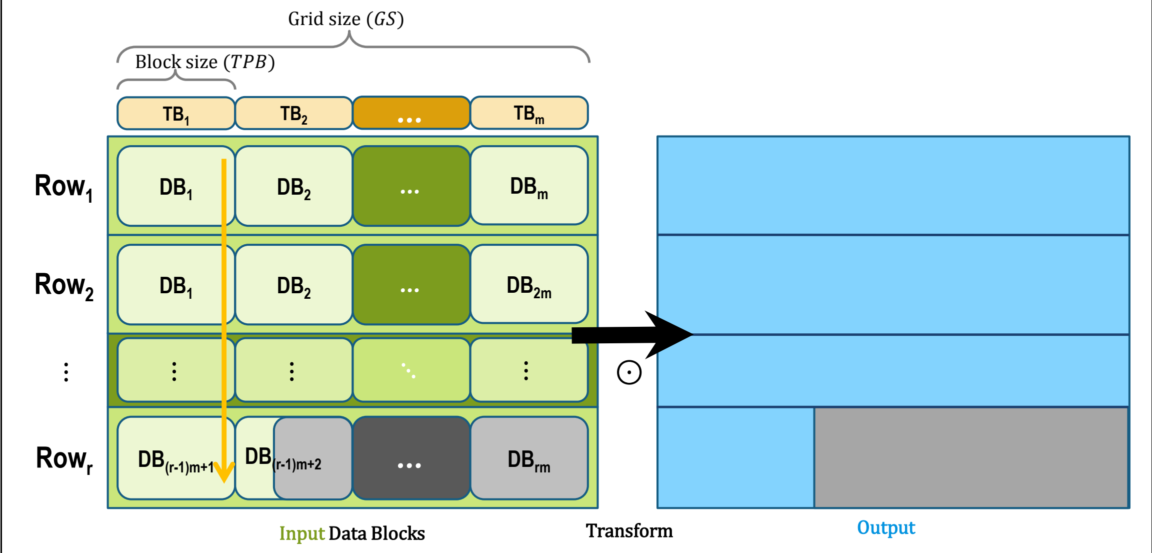


Figure 5.12: *Column* DASk layout. The input dataset (*n*) is divided into *m* fixed size data blocks. The *m* data blocks are tiled into a 2D grid with *C* fixed-size columns and *R* rows per column (which varies as needed to cover all data). Each thread block is assigned one data column and is responsible for processing all data blocks along its assigned data column. The last data block in each data column may require careful range checking, but the first *R*-1 data blocks require no range checking.

Since the *C* and DBS are both fixed, then the size of one data row is equal to *rowSize*=*C*·*DBS*. The number of grid rows (*R*) can be computed as *R* = (*n*+(*rowSize*-1))/*rowSize*. I assign one thread block to each data column and then have all the column thread blocks cooperatively stride through the entire data set one row at a time (*stride* = *rowSize*). Similar to the way I coded the Block DASk, I use template parameters, multiple work items per thread, amortized range checking (across rows then within data blocks), and software pipelining for the Column DASk to achieve better performance.

5.3.1 Column DASk Code

A simplified version of the Column DASk for the Copy primitive is shown in Figure 5.13. My implementation of the Column DASk uses a while loop to iterate over all the full rows that require no range checking, requiring the usual loop overhead. Recall that the last row may only be partially full, which implies range checking. Consequently, I repeat the amortized range check pattern from the Block DASk here (see Section 5.1.2). This results in four shaded boxes that need to be filled in by the GPU programmer to adapt the Column DASk for their own algorithm. The code in the first two shaded boxes are the same since both require no range checking. The code in the third shaded box deliberately does nothing (as the entire data block is out-of-range), and the code in the fourth shaded box basically repeats the code in the first shaded box but with more careful range checking. Just to demonstrate its usage, I use the Block by Block BASk for mapping thread warps onto data inside each data block, in my own actual code, I use the Warp by Warp BASk instead. My Histogram case study in Chapter 8 uses the Column DASk as its underlying framework.

```
template < valT, logWarpSize, TBS, GridCols, nWork >
__global__ void Copy_Column( D, S, start, stop ) {
  // Compiler-Time Variables (become constants at run-time)
  const U32 DBS     = nWork*TBS;            // Data Block Size (512 = 4*128)
  const U32 TPR     = BlockSize * GridCols; // Threads per Row (across grid)
  const U32 rowSize = nWork * TPR;          // Data per Row (across grid)

  // Map CTA parameters onto data offsets
  U32 bid = (blockIdx.y * gridDim.x) + blockIdx.x;  // Block ID (bid)
  U32 tid = threadIdx.x;                   // Thread ID (tid)
  U32 dataOff = (bid * DBS) + tid;         // starting data offset

  // Compute Row Info
  U32 nElems = stop - start + 1;
  U32 nFullRows  = nElems / rowSize;
  U32 nFullElems = nFullRows * rowSize;
  U32 nLeftOver  = nElems - nFullElems;
  U32 startIdx = start + dataOff;
  U32 stopIdx  = (nFullRows * rowSize) + startIdx;
  U32 currIdx  = startIdx;

  // Get start I/O pointers
  const U32 * in  = &(S[currIdx]);
        U32 * out = &(D[currIdx]);

  // Process all full rows (*NO* range checking needed)
  while (currIdx < stopIdx) {
     // #1) Copy Data Block (copies entire data row across all blocks)
     valT v1, v2, v3, v4;

     // Load values
     if (nWork >=  1u) { v1 = in[(0u*TBS)]; }
     if (nWork >=  2u) { v2 = in[(1u*TBS)]; }
     if (nWork >=  3u) { v3 = in[(2u*TBS)]; }
     if (nWork >=  4u) { v4 = in[(3u*TBS)]; }

     // Store values
     if (nWork >=  1u) { out[(0u*TBS)] = v1; }
     if (nWork >=  2u) { out[(1u*TBS)] = v2; }
     if (nWork >=  3u) { out[(2u*TBS)] = v3; }
     if (nWork >=  4u) { out[(3u*TBS)] = v4; }

    // Move to next data row
    currIdx += rowSize;
    in  += rowSize;
    out += rowSize;
  } // end while

  if (nLeftOver) { // Leftover row (with range checking)
    // Compute Last Row & Data Block Info
    U32 nSkipFull  = nFullRows * rowSize;   // Skip past full rows
    U32 startIdx   = start + nSkipFull + dataOff;
    U32 currIdx    = startIdx + tid;        // Get data offset
    U32 blockOff   = (bid * DBS) + start;   // starting block offset
    U32 blockStart = start + blockOff;
    U32 blockStop  = blockStop + DB_LAST;

    const U32 * in  = &(S[currIdx]);
          U32 * out = &(D[currIdx]);

     // *data block range checks* against [start, stop]
     bool inRange  = (blockStop <= stop);
     bool outRange = (stop < blockStart);
     if (inRange) {
        // #2) In Range, copy data block (with *NO* range checking)
        ...  // <Code is the same as shaded box #1 above>

    } else if (outRange) {

        // #3) Out of Range, Exit Kernel (nothing to do)

    } else {

        // #4) Overlaps Case, process block (with careful range checking)
        U32  D1, D2, D3, D4;
        bool T1, T2, T3, T4;
        valT v1, v2, v3, v4;
        // Get data offsets
        if (nWork >= 1u) { D1 = (0u*TBS) + currIdx; }
        if (nWork >= 2u) { D2 = (1u*TBS) + currIdx; }
        if (nWork >= 3u) { D3 = (2u*TBS) + currIdx; }
        if (nWork >= 4u) { D4 = (3u*TBS) + currIdx; }
        // Range Check (each data access)
        if (nWork >= 1u) { T1 = (D1 <= stop; }
        if (nWork >= 2u) { T2 = (D2 <= stop; }
        if (nWork >= 3u) { T3 = (D3 <= stop; }
        if (nWork >= 4u) { T4 = (D4 <= stop; }
        // Load values (with range checking)
        if (nWork >=  1u) { if (T1) { v1 = in[( 0u*TBS)]; } }
        if (nWork >=  2u) { if (T2) { v2 = in[( 1u*TBS)]; } }
        if (nWork >=  3u) { if (T3) { v3 = in[( 2u*TBS)]; } }
        if (nWork >=  4u) { if (T4) { v4 = in[( 3u*TBS)]; } }
        // Store values (with range checking)
        if (nWork >=  1u) { if (T1) { out[( 0u*TBS)] = v1; } }
        if (nWork >=  2u) { if (T1) { out[( 1u*TBS)] = v2; } }
        if (nWork >=  3u) { if (T1) { out[( 2u*TBS)] = v3; } }
        if (nWork >=  4u) { if (T1) { out[( 3u*TBS)] = v4; } }

    } // end *data block range checks*
  } // end *Left Over Row*
} // end Copy_Column
```

Figure 5.13: My *Column* DASk for the Copy primitive. The code outside the four shaded boxes makes up the scaffolding for data access using the Column DASk. The code inside the four shaded boxes (#1-#4) would be replaced by code to support your own algorithm. All the full rows require no range checking, the left-over partial row repeats the amortized range checking pattern from the Block DASk. The lighter grey `if {…}` wrapper statements get elided away at compile time.

5.3.2 Column DASk Conclusion

My Column DASk has several benefits and several limitations that I discuss next.

Three main Benefits:

- Supports proper range checking of all data [*start*, *stop*] while only doing range checking on the last row of data. The cost of doing the range check on the last row of data is amortized across the other data blocks in the column.
- Supports an efficient access pattern into memory using a 2D grid layout.
- Works well when accumulating data in a dataset but the order in which the data is accumulated does not matter.

Five main Disadvantages:

- The GPU programmer needs to write two slightly different version of the same code to handle the three different range check cases {in-range, out-of-range, overlap}. This can also lead to cut and paste errors, if the programmer is not careful when fixing bugs.
- Increased register pressure to set up the range check variables used by this DASk.
- Extra branching to handle looping and range checks. In particular the while loop used to loop over the full rows of data gets called once per data block. Fortunately, my DASk implementation was written to avoid divergent branching.
- Performance sensitivity to the initial grid size. If the chosen grid size by the programmer is not an even multiple of the actual workLoad (#SMs * #Blocks), then performance suffers as some SMs process a few extra rows while the rest of the SMs idle.
- The cooperative stride across all data blocks results in each new data block being a full data row apart from the previous data block which is not well localized and hinders caching by the memory controllers when moving between data blocks.

5.4 *Row* DASk

The *Row* DASk is also uses a two dimensional grid layout similar to the Column DASk but with the rows and columns reversed. For the Row DASk, the original input data (*n*) is divided into *m* fixed-size data blocks (*DBS*) with the data blocks being laid out on a 2D grid, with the total number of blocks (*m*) needed to cover data computed as $m = \lceil n/DBS \rceil$. The number of rows is chosen ahead of time by the GPU programmer as a static fixed size constant (*R*= *grid.height*). The number of columns (*C* = data blocks per row) is allowed to vary with *n* in order to fully over all input data and can be initially computed as $C = \lceil m/R \rceil$, however, as will be seen shortly to better support load balancing of work across thread blocks, my actual computation of the number of columns per row is more complex. The Row DASk maps one thread block onto each data row, with each thread block being responsible for processing all data blocks along its assigned row.

As shown in Figure 5.14, the Row DASk is similar to the Column DASk. There are several main differences between these two DASks: layout, alignment, range checking, and load balancing. To better support coalescence, I warp align the input data to a data warp boundary before marching down the first row (The Block and Column DASks do not warp align data, so performance may suffer if the user passes in unaligned data arrays). Supporting warp-aligned data implies that the first data warp in the first data block in the first row may be partially empty, requiring range checking. Similarly, one of the data blocks in the last row may also be partially empty and also may requiring careful range checking as well.

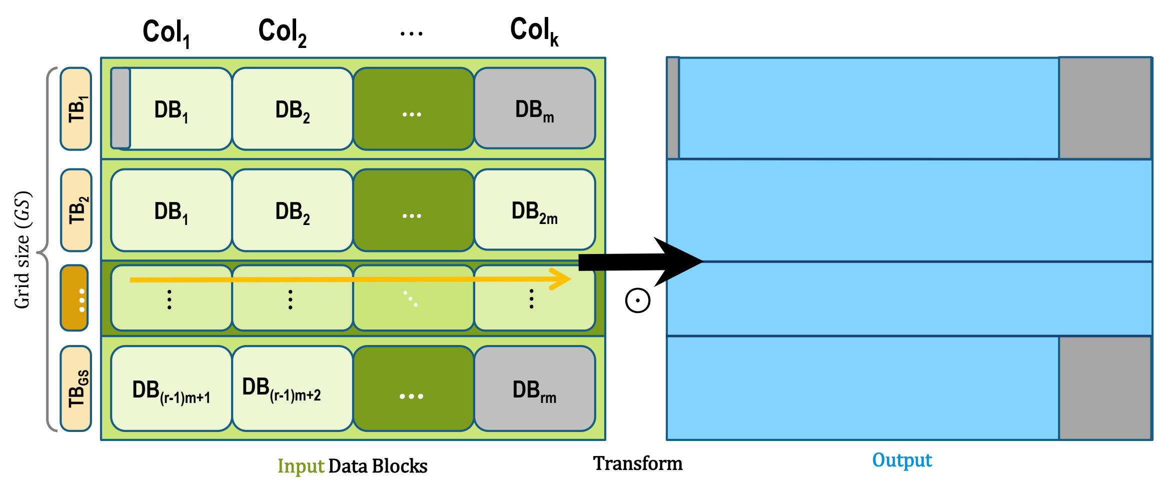


Figure 5.14: *Row* DASk layout

Returning to the idea of load-balancing, consider a simple but naïve load-balancing scheme, where all rows but the last row would be full with the last row being only partially full. The thread block assigned to this last row would complete much more quickly than the other thread blocks assigned to full rows underutilizing our processing resources. To help avoid underutilization, I have created a more complex load-balancing scheme to distribute the data blocks approximately equally across all the thread blocks representing data rows. This results in a difference of at most one data block of work between any two thread block on their assigned rows. Because, range checking is more expensive than not range checking, I seek to avoid assigning extra data blocks to rows that require range checking. The first row may require some range checking on its first data warp, and one other thread block (row) may have the last partially full data block in its row (typically the last data row but not always), which also requires range checking. As a result, I have written my load balancing code to deliberately bias against assigning a left-over data block to either the first or last row (meaning the thread block with the last data block in its row) unless there is no other choice (in other words, *m* is divided evenly by *R*).

I use the Row DASk as the underlying framework for my case studies for *Scan* (Chapter 6) and *RadixSort* (Chapter 9). In the next few sub-sections, I show how I actually warp align my data (Section 5.4.1), introduce a new range checking pattern (Section 5.4.2), and load balance data blocks across thread blocks (Section 5.4.3).

5.4.1 Warp Alignment

Experienced CPU programmers know that aligning memory accesses to start on cache-line boundaries (0, 64, 128, 192, or 256 bytes) makes overall memory throughput faster. There is a similar throughput advantage when aligning global memory accesses on GPUs to warp-line boundaries (128 bytes). To align a data pointer or index to a warp line boundary, I find the first multiple of 128 bytes that starts just before the requested starting pointer or index -- for instance, `alignIdx = (idx/128)*128`. Since 128 is a power of two, the same result can be achieved using less expensive bit masking: `alignIdx = idx & ~127.` Here is the snippet of code (see Figure 5.15) I use to align 32-bit data elements to a warp-line memory boundary:

```
// Compiler Variables
U32 WarpSize = 1 << logWarpSize; // 32 threads per warp
U32 WarpMask = WarpSize – 1;     // 31 = 0x1F = bitmask<00011111>
...
// Align <src, dest> to a warp boundary
U32 src_Pad = start & WarpMask;         // (start % WarpSize)
U32 src_AlignPos = start & ~WarpMask;   // (start/WarpSize)*WarpSize
U32 dst_AlignPos = startDest – src_Pad;
```

Figure 5.15: Warp Aligning Data Access.

Warp aligning data (see Figure 5.15) access may result in some threads accessing memory locations before the true data start. Careful range checking on the first data warp (or first data block) is therefore essential to avoid out of range access errors.

5.4.2 ‹FIRST?› ‹MIDDLE*›‹LAST?› Range Check Pattern:

Range checking for the Row DASk is sufficiently different from the previous DASks that I created a new that I call the ‹FIRST?› ‹MIDDLE*› ‹LAST? › or ‹BOTH› range check pattern to avoid out-of-range memory accesses. The goal here is to push any range checking out to the ‹FIRST?› and ‹LAST?› data blocks leaving the large ‹MIDDLE*› section of data blocks without a need to do any range checking. I use regular expression notation to indicate how many data blocks are involved in each group, where the question mark (?) means zero or one data blocks, and the asterisk means (*) means zero or more data blocks.

I assume all input, and output, data is located in a range [*start*, *stop*]. So, lets briefly discuss the range check requirements of each of the four groups.

- ‹FIRST?›: As a result of warp-aligning the input data to a warp boundary, the first data block may require partial [*start*, …) range checking when accessing data elements in the first data warp. If the data already was passed in warp aligned (meaning the start parameter on a multiple of the WarpSize, i.e. [0, 32, 64, …]), then there is no need to range check the first data block.
- ‹LAST?›: As a result of using fixed size data blocks, the last data block most likely is only partially full and thus may require partial (…, *stop*] range checking when accessing data elements in this last block. If the ‘stop’ parameter happens to fall exactly on a multiple of the fixed block size (DBS) after accounting for warp-alignment, then there is no need to range check the last data block.
- ‹MIDDLE*›: All the other in-between data blocks require no range checking.
- ‹BOTH?›:I include this case for completeness. It is possible that the input dataset is small enough so that both the ‹FIRST? › and ‹LAST?› data blocks are one and the same. This case requires full [*start*, *stop*] range checking unless the dataset is both warp-aligned and this single data block contains exactly DBS elements.

Given enough data, any range checks for both the ‹FIRST?› and ‹LAST?› blocks are amortized across the rest of the ‹MIDDLE*› data blocks. Even though the abstract concept of range checking is orthogonal from other concepts like warp alignment, load balancing and CTA layout, my actual DASk code intermingles these concepts.

The code for the <FIRST?> <MIDDLE*> <LAST?> or <BOTH?> range checking pattern is shown in Figure 5.16. The shaded boxes represent the GPU programmer's code that would be replaced for other algorithms while still range checking.

```
// Get Alignment Info
...
// Load Balance (Data blocks across rows (thread blocks))
...
// Get Row Info
U32 IsStartAligned = ((start == src_AlignPos) ? 1u : 0u);
U32 src_StartPos = (rowBlockStart * DBS) + src_AlignPos;
U32 dst_StartPos = (rowBlockStart * DBS) + dst_AlignPos;
U32 isFirstRow   = (rowBlockStart == 0u);
U32 src_StopPos  = (rowBlockCount * DBS) + src_StartPos;
// Range Check Setup
U32 rcFirst = (isFirstRow && !IsStartAligned);
U32 rcLast  = isLastRow & (!isStopAligned);
U32 rcBoth  = isFirst & rcLast & singleDataBlockRow;
...
if (rcBoth) { // <BOTH>
    // #1) Copy Data block (range check on [start, stop])
    d1 = ...;
    d2 = ...;
    ...
    bool T1 = (start <= d1) & (d1 <= stop); // Range Checks
    bool T2 = (start <= d2) & (d2 <= stop);
    ...
    if (nWork>=1) { if (T1) { v1 = in[d1];  }} // Load values
    if (nWork>=2) { if (T2) { v2 = in[d2];  }}
    ...
    if (nWork>=1) { if (T1) { out[d1] = v1; }} // Store Values
    if (nWork>=2) { if (T2) { out[d2] = v2; }}
    ...
} // end <BOTH>

if (rcFirst) { // <FIRST>
    // #2) Copy Data Block (range check on [start, ...) for data accesses)
    d1 = ...;
    ...
    bool T1 = (start <= d1); // Range Checks
    ...
    if (nWork>=1) { if (T1) { v1 = in[d1];  }} // Load values
    ...
    if (nWork>=1) { if (T1) { out[d1] = v1; }} // Store Values
    ...
} // end <FIRST>
while (src_Base < src_StopPos) { // <MIDDLE>
```

```
    // #3) Copy Data Block (*NO* range checks)
    d1 = ...;
    d2 = ...;
    ...
    if (nWork>=1) { v1 = in[d1];  } // Load values
    if (nWork>=2) { v2 = in[d2];  }
    ...
    if (nWork>=1) { out[d1] = v1; } // Store Values
    if (nWork>=2) { out[d2] = v2; }
    ...

  // Move to next data block (along row)
  src_Base += DBS;
  dst_Base += DBS;
  in  += DBS;
  out += DBS;
} // end <MIDDLE>
if (rcLast) { // <LAST>
    // #4) Copy Data Block (range check on (..., stop] for data accesses)
    d1 = ...;
    ...
    bool T1 = (d1 <= stop); // Range Checks
    ...
    if (nWork>=1) { if (T1) { v1 = in[d1];  }} // Load values
    ...
    if (nWork>=1) { if (T1) { out[d1] = v1; }} // Store Values
    ...
} // end <LAST>
```

Figure 5.16: A quick sketch of my *Row* DASk with the primary focus on the ‹FIRST?› ‹MIDDLE*› ‹LAST? › or ‹BOTH› range check pattern. The code outside the four shaded boxes makes up the scaffolding for data access. The code in the four shaded boxes (#1-#4) would be replaced by GPU Programmers own code. The third shaded box corresponds to the ‹MIDDLE*› pattern that requires no range checking, the other three shaded boxes correspond to the ‹BOTH›, ‹FIRST? ›, and ‹LAST? › groups that require range checking over the ranges [*start*, *stop*] or [*start*, …) or (…, *stop*] respectively to avoid data access errors.

5.4.3 Load Balancing

To load-balance data blocks across rows, I conceptually use a 2D grid of data rows (one row per thread block) that covers all the data blocks (see Table 5.3). I categorize the rows into three types: First Row, Middle Row, and Last Row.

	Column Types (Data Blocks)	
Row Types	**Full**	**Left Over**
`<First Row?>`	`<First Block?><Blocks*>`	`<Extra Block?>`
`<Middle Rows*>`	`<Blocks*>`	`<Extra Block?>`
`<Last Row?>`	`<Blocks*><Last Block?>`	`N/A`

Table 5.3: Load balancing data blocks across rows (thread blocks) for *Row* DASk.

The First Row is the only row that can have a First Block. This first data block may require [*start*, …) range checking as a result of warp-alignment. The Last Row is the only row that may have a Last Block. This last block most likely will require (…, *stop*] range checking as a result of using fixed size data blocks. All other data blocks require no range checking because they are guaranteed to be fully contained in the range [*start*, *stop*], which implies that all Middle Rows require no range checking as well. Each row can optionally have one extra data block {0|1}, this extra data block is used to spread out any left-over data blocks across all the rows (thread blocks) as evenly as possible. Since the first row and last row may do more work overall as a result of range checking than the other middle rows, my load-balancing code biases against adding an extra data block into the first and last rows unless there is no other choice. The last row only gets an extra data block if and only if the number of rows (R) divides evenly into the number of data blocks (m), meaning that $0 = \text{mod}(m, R)$. Similarly, the first row only gets an extra data block if and only if the number of left-over extra blocks is equal to the number of rows (R) minus one.

As shown in Figure 5.17, my code for load balancing works as follows:

1) Get total number of data blocks.
2) Divide data blocks evenly across rows.
3) Compute any extra data blocks.
4) Distribute the extra blocks across as many rows (thread blocks) as needed to cover them all, biasing against adding extra blocks to the first and last rows.

```
__device__ __forceinline__
void LoadBalance
(
  U32 & rowBlockCount,    // OUT – data blocks assigned to this row
  U32 & rowBlockStart,    // OUT - starting block for this row
  U32 nRows,              // IN - nRows
  U32 nBlocks,            // IN – Total Blocks
  U32 currRow             // IN – current Row represented by this thread block
)
{
  // Compute 'Blocks per Row'
  U32 nFullBPR = nBlocks / nRows;
  U32 leftOver = nBlocks - (nFullBPR * nRows);

  // Compute number of rows & row length
  // for each section <First><Full><Extra>
  U32 blockInFirstRow  = nFullBPR;
  U32 blocksInPartRows = nFullBPR;
  U32 extra     = ((leftOver == 0) ? 0 : 1u);
  U32 nFullRows = ((leftOver == 0) ? nRows : leftOver);
  U32 nPartRows = nRows - nFullRows;
  U32 blocksInFullRows = nFullBPR + extra;

  // Is it safe to bias against the first row ?
  U32 nFirstRow = 0u;
  if (nPartRows >= 2u) {
    if (nFullCPR > 0u) {
       nFirstRow  = 1u;
       nPartRows -= 1u;
    }
  }

  // Get Row Counts & Row Starts
  U32 rowCount = 0u;
  U32 startSum = 0u;

  // Move from 'zero'-based to 'one'-based indexing
  currRow = currRow + 1u;

  // Process first row section
  if (nFirstRow > 0u) {
    // First Row section
    rowCount  = blocksInFirstRow;
    currRow  -= 1u;
  }

  // Process 'full' row section
  if (currRow > 0u) {
    U32 fullCnt = ((currRow >= nFullRows) ? nFullRows : currRow);
    startSum = rowCount + ((fullCnt - 1u) * blocksInFullRows);
    rowCount = blocksInFullRows;
    currRow -= fullCnt;
  }
  // Process 'Partial' row section
  if (currRow > 0u) {
    // Part Row Section
    rowCount  = blocksInPartRows;
    startSum += blocksInFullRows             // Add in last row in full section
              + ((currRow - 1u) * blocksInPartRows);    // Add in partial rows
  }
  // Output 'Load Balanced' row count & row starts
  rowBlockCount = rowCount;
  rowBlockStart = startSum;
} // end LoadBalance
```

Figure 5.17: The LoadBalance code for my *Row* DASk. The code works in 3 sections as 1) compute data blocks in first row, 2) compute blocks in full rows, 3) compute blocks in partial rows.

To compute the starting block for each row, I basically add up row counts for up to three groups {*first*, *full*, *partial*}. First, I determine the number of blocks in the first row. Second, I determine the number of full rows (rows that receive an extra block), Third, I determine the number of partial rows (rows without an extra block). Next, I determine for the current row whether it is the first row, in the second group of full rows, or in the third group of partial rows. Then I add up the contributions from each group that precedes the current row or the partial contribution of the group that the current row is in to arrive at the starting offset. The group type that the current row belongs to also determines the number of data blocks for this row.

As a simple example, assume that there are five rows and 23 data blocks. Distributing the data blocks across the rows results in four full data-blocks per row (4 = $\lceil 23/5 \rceil$) plus three extra data blocks (3 = mod(23, 5)). Since there are fewer than 4 (5-1) data blocks, I don't put an extra block in the first row, and my full group will be three rows long, my partial group will be one row long 1 = 2-1. So, my load balancing code would assign ‹4,5,5,5,4› data blocks, respectively, to each of the five rows.

As can be seen in Figure 5.17, the load-balancing code is complicated with lots of divisions, multiplications, and RAW dependencies between instructions. Despite the many branches in the code, branch divergence does not become a significant factor because all the threads in each warp belong to the same thread block (or data row). They therefore each follow the same branch path through this function. The load balancing code does result in extra overhead, however this cost is amortized across all the data blocks assigned to the row {*R* | *R*-1}.

5.4.4 Row DASk conclusion

My Row DASk has several benefits and several limitations that I discuss next.

Five main Benefits:

- To better support coalescence, this DASk aligns the starting warp of the input dataset to a warp boundary [0, 32, 64, …].
- For better processor utilization, this DASk load balances work as evenly as possible across thread blocks (rows).
- Supports proper range checking of all data [*start*, *stop*] while pushing range checks into the first and last data blocks, thus allowing the range check costs of at most one data block to be amortized across all R data blocks in each row.
- This DASk can support a sequential ordering for the data parallel algorithm as each thread block marches along its data row, block by block. The GPU programmer still needs to preserve a sequential order for data blocks when accessing short runs of data by each thread in the thread block (See my *Reduce* and *Scan* case study in Chapter 6 for a good example of one way to handle this).
- Supports an efficient access pattern into memory using a 2D grid layout.

Five main Disadvantages:

- High setup costs (warp alignment, load balancing, range checking, …) that need to be amortized across lots of data blocks in each row.
- The GPU programmer needs to write four slight different version of the same code to handle the four different range check cases {Both, First, Middle, Last}. This can also lead to cut and paste errors, if the programmer is not careful when fixing bugs.
- Increased register pressure to set up and use the load balancing and range check variables used by this DASk.
- Extra branches to handle the four range check cases {*BOTH*, *FIRST*, *MIDDLE*, *LAST*}. In particular the while loop used for the ‹MIDDLE*› case gets called once per data block. Fortunately, my DASk implementation was written to avoid divergent branching.
- Performance sensitivity to the initial grid size. If the chosen grid size by the programmer is not an even multiple of the actual workLoad (*workLoad* = *nSMs* × *nConBlocks*), then performance suffers as some SMs process a few extra rows while the rest of the SMs idle. This is actually a problem with the Block DASk as well but that DASk tends to launch thousands of thread blocks so the problem effectively gets amortized away. The Grid size for this DASk tends to be a lot smaller so the problem is much more noticeable.

5.5 Lessons Learned from this Copy case study

There are several lessons learned from both Chapter 4 and Chapter 5.

- **Use both ILP + TLP for better performance**
    - For better ILP
        - Support multiple work items per thread (*nWork*=4).
        - To reduce register pressure, batch work items in groups of 4.
        - Group k similar instructions from *k* work items to reduce RAW dependencies between instructions.
        - Prefer manual over automatic loop unrolling.
        - 2-4 work items per thread is a good starting point.
    - For better TLP
        - For block size, 128 threads is a good starting point (*TBS* = 128).
        - For grid size, pick a multiple of the work load (*workLoad* = *nSMs* × *nConBlocks*) as a good starting point (*GS* = 224 = 14 SMXs * 16 concurrent blocks on GTX Titan).
    - Prefer increasing TLP over increasing ILP.
        - Increasing TLP resulted in up to 2.85× faster performance.
        - Increasing ILP resulted in up to 1.09× faster performance.
        - These approaches are orthogonal, so increase both TLP and ILP for best performance.
- **Amortize setup costs across multiple work items for better performance**
    - Range check once per thread instead of once per work-item
    - Setup pointers once per thread instead of once per work-item.
- **Understand GPU memory for better performance**
    - Registers are faster than shared memory which is faster than global memory which is faster than CPU RAM.
    - Respect coalescence for peak throughput. Access 32-bit data elements in global memory using a warp-aligned, warp-sequential fully used data access pattern.
    - For better performance, cluster similar memory accesses together in small batches of *k* work items (*k* loads followed by *k* stores). This results in better performance than interleaving memory accesses (in, out, in, out) from different memory arrays or types of memory.

Better Cluster similar memory accesses into batches	**Worse** Interleave different memory accesses

`v1=D[w1]; v2=D[w2]; v3=D[w3]; // In batch`
`...`
`S[w1]=v1; S[w2]=v2; S[w3]=v3; // Out batch`
	`D[w1]=S[w1]; // Input & Output`
	`D[w2]=S[w2]; // ditto`
	`D[w3]=S[w3]; // ditto`

Table 5.4: Batching vs. Interleaving Memory access.

- **Use Block Access Skeletons (BASks) to jump start efficient memory access within each data block.**

  - *Block by Block* BASk
    - *Pros*: Supports coalescence, less setup, easier to code, easier to reason about
    - *Cons*: Ping-Pong access pattern is slightly less localized, may require more barrier synchronization.
  - *Warp by Warp* BASk
    - *Pros*: Supports coalescence, slightly more localized sequential access pattern, Warps can start work independently, may require less barrier synchronization.
    - *Cons*: More indexing setup, harder to code, harder to reason about.
- **Use Data Access Skeletons (DASks) to jump start efficient memory access across all elements in a large data set.**
  - *Pros*:
    - Supports Genericity (general data types, warp sizes, CTA layouts, …)
    - Framework code is already written, tested, and working
    - Framework code uses efficient memory access patterns.
    - Body code (shaded boxes) can be replaced by the GPU programmer for specific problem spaces.
    - Supports experiments on both ILP and TLP to find the best performing configuration.
    - TLP is supported directly by framework.
  - *Cons*:
    - More complex than simple kernels.
    - Similar code across multiple shaded boxes can result in cut and paste errors.
    - ILP is mostly controlled by GPU programmer as manual loop unrolling to support multiple work items must be implemented directly within each shaded box.
    - Higher register pressure may result from setup overhead and range checking.
  - Block DASk:
    - *Pros*: Simplest DASk
    - *Cons*: Assumes input is already warp-aligned. Requires range checking on each data block.
    - *Example*: See *Copy* (this Chapter).
  - Column DASk:
    - *Pros*: Only requires range checking on last partially full data row.
    - *Cons*: Assumes input is already warp-aligned, memory access pattern is less localized (all thread blocks stride across one data row). Small grid sizes result in poor performance when grid size is not evenly divisible by the workLoad.
    - Example: See *Histogram* (Chapter 8).
  - Row DASk:

- *Pros*: Automatically warp-aligns input data, supports load balancing across thread blocks. Supports access patterns where order matters. Only requires range checking on first and last data blocks.
- *Cons*: High startup overhead (load balancing and range checking). Small grid sizes results in poor performance when grid size is not evenly divisible by the workLoad.
- *Example*: See *Scan* (Chapter 6), *Radix Sort* (Chapter 9).

6.0 Case Study: *Reduce* and *Scan* on the GPU

In this chapter, I demonstrate my Row data access skeleton (DASk) on two useful parallel primitives, Reduce and Scan. The Reduce primitive produces a total sum by accumulating *n* input elements into a single final sum. The terms "sum" and "accumulate" refer not only to addition but to any associative operation, such as multiplication, maximum, or average. The name "Reduce" comes from Map/Reduce, which is a popular paradigm for distributed computing at very large scales. The Scan primitive, sometimes called Prefix Sum, produces a running sum by accumulating *n* input elements into *n* output elements, where the i^{th} output element is either the *inclusive* {1 ... *i*} or *exclusive* {1 ... *i*-1} prefix sum of the first *i* input elements.

I have chosen to base both my Reduce and Scan primitives only on the associative property $\{a+(b+c) = (a+b)+c\}$, so that I can also support non-commutative "summation-like" operations, such as matrix multiplication. My algorithms may regroup, but not reorder; all sub-problems must work with sequential runs of consecutive data.

To support the GPU's 2-level cooperative thread array (CTA), I think of data as sequential blocks containing sequential runs. Since the CTA is 2-level, I use a 2-part solution. 1) The Row DASk at CTA level 1 coordinates thread blocks within a grid for sequential access to data blocks. 2) Using a similar mechanism at CTA level 2 coordinates individual threads within a block for sequential access within each data block. To keep throughput high in my implementations, I use both thread-level parallelism (TLP) and instruction-level parallelism (ILP). To support experiments on TLP and ILP, I parameterize my kernels by the number of warps per thread block ‹*nWarps*› and by the number of work items per thread ‹*nWork*›.

In this case study, three main issues hinder throughput:

1) Finding a global memory access pattern that supports both coalescence (*stride* = 32) and sequential access (*stride* = 1). Both access patterns use different strides and therefore are mutually exclusive.
2) Avoiding serialized instruction replays caused by *k*-way bank conflicts when accessing shared memory.
3) Avoiding reduced TLP caused by various resource constraints on occupancy.

6.1 Introduction

***Reduce* primitive** Computes the total *sum* of a sequence *A* containing *n* elements.
Input: binary associative operator $\oplus$ with identity $\mathbb{I}$, and sequence $A = [a_1, a_2, \cdots, a_n]$,
Output: $sum = \mathbb{I} \oplus a_1 \oplus a_1 \oplus \cdots \oplus a_n$ where *sum* is a singleton result.

***Scan* primitive** creates the *prefix sum* of a sequence *A* containing *n* elements.
Input: A binary associative operator $\oplus$ with identity $\mathbb{I}$, and sequence $A = [a_1, a_2, \cdots, a_n]$,
Output: A scanned sequence $S = [s_1, s_2, \cdots, s_n]$,
where $s_i = \mathbb{I} \oplus a_1 \oplus a_2 \oplus \cdots \oplus a_{i-1}$ for *exclusive scan*
or $s_i = a_1 \oplus a_2 \oplus \cdots \oplus a_i$ for *inclusive scan*.

Reduce and Scan primitives are basic building blocks for many parallel algorithms (Blelloch 1989 and 1990; Blelloch and Maggs, 1996; Hillis and Steele, 1986). Each takes as input an associative, but not necessarily commutative, binary summation operator $\oplus$, its identity $\mathbb{I}$, and a sequence $A = [a_1, a_2, \ldots, a_n]$. The Reduce primitive (Harris and Sutherland, 2003), also known as "Fold" or "Total-sum", returns *sum* = $\mathbb{I} \oplus a_1 \oplus a_2 \oplus \cdots \oplus a_n$. The Scan primitive (Blelloch, 1989), also known as "Prefix-sum", returns either an inclusive prefix-sum sequence, in which $s_1 = a_1$ and, for $i > 1$, $s_i = s_{i-1} \oplus a_i$, or an exclusive prefix-sum sequence, in which $s_1 = \mathbb{I}$ and, for $i > 1$, $s_i = s_{i-1} \oplus a_{i-1}$.

In my own GPU algorithms, I use Reduce to produce summation results for performance metrics, such as minimums, maximums, totals, averages. I use Scan to implement data partitioning schemes like counting sort. Scanning can count data elements to determine starting locations so that each individual thread knows where to safely access its assigned data without competing for access with other concurrently running parallel threads.

As Table 6.1 shows the binary operator $\oplus$ can represent not only addition, but also product, max, min, "and," and "or". I assume that $\oplus$ is associative $\{(a \oplus b) \oplus c = a \oplus (b \oplus c)\}$, but not commutative

$\{(a \oplus b) = (b \oplus a)\}$. The fact that data can be regrouped but not reordered complicates the GPU implementations for both block and thread levels of the CTA hierarchy.

Operator $\{\oplus\}$	**Identity** $\{\mathbb{I}\}$	**Math**	**Code**
Sum $\{+\}$	0	$c = a + b$	`c=a+b;`
Floating point** Sum $\{+\}$	0.0	$c = fl(a + b)$	`c=a+b;`**
Product $\{\times\}$	1	$c = a \times b$	`c=a*b;`
Floating point** Product $\{\times\}$	1.0	$c = fl(a \times b)$	`c=a*b;`**
Matrix Multiply $\{\times\}$ *Non-commutative*	$\boldsymbol{I}$	$C = AB$	3-level nested loop
Minimum $\{min\}$	$+\infty$	$c = \begin{cases} a & if\ a \leq b \\ b & if\ b < a \end{cases}$	`c=(a<=b)?a:b;`
Maximum $\{max\}$	$-\infty$	$c = \begin{cases} a & if\ a \geq b \\ b & if\ b > a \end{cases}$	`c=(a>=b)?a:b;`
Logical AND $\{\wedge\}$	*True*	$c = a \wedge b$	`c=a&b;`
Logical OR $\{\vee\}$	*False*	$c = a \vee b$	`c=a|b;`

Table 6.1: Common Reduce and Scan binary operators. **Floating point operators are not fully-associative due to truncation and round-off errors.

Floating Point Associativity: Programmers should be aware that floating point arithmetic operators are not fully associative since outputs have errors as floating point operations truncate and round-off the results to fit within the fixed-size data types[1] (Press et al, 2007). As a result, CPU serial primitives and GPU parallel primitives on floating point data runs can give slightly different sums. Reordering and regrouping on long runs of data containing large variations in exponents can change outputs. One possible advantage of forbidding commutativity (reordering) and relying on associativity (regrouping) alone is that floating point evaluations are a bit more stable[2].

[1] Floating point issues are described in much greater detail in the book "Numerical Recipes …" (Press et al, 20007).
[2] There are techniques that provide even more stability, such as storing values in a min heap on absolute value and repeatedly summing the two values nearest zero. Unfortunately this approach has too many data dependencies to work well on GPUs.

Overflow: Programmers should ensure that sums (especially of long data runs) do not overflow the underlying data type's maximal (or minimal) value. For instance, adding a billion 32-bit unsigned integers may require storing intermediate and final sums as 64-bit unsigned integers.

Diagrams: Data organization and access patterns are necessary to improve Reduce and Scan performance, and these are often easier to grasp from diagrams. Table 6.2 introduces symbols that I use in my diagrams throughout this chapter.

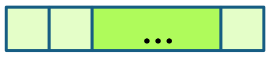
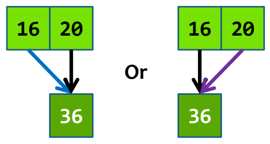
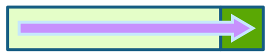
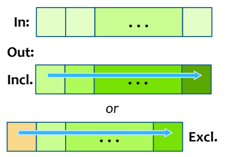
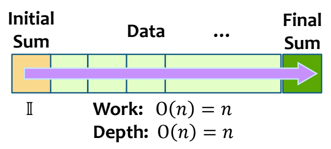
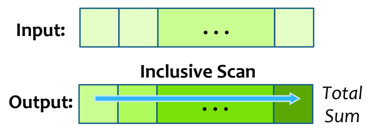
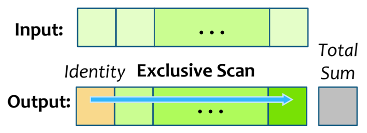
Name	Abbr.	Description	Symbol
Run		An input sequence of *n* data elements.	
Sum	⊕	An associative binary operation that accumulates two inputs into an output. $c = a \oplus b$. Arrow color indicates access to the current entry (black) and reach back (blue) or reach forward (purple) one or more entries.	
Identity	𝕀	The identity element for operator ⊕, i.e. $a = \mathbb{I} \oplus a$ for all $a \in \mathbb{U}$ (zero for addition, one for multiplication).	
Serial Reduce	SR*n*	In serial, reduce an input short run of size $n \in [2\text{-}32]$ into a final-sum output. ~One I/O per element, plus one for output.	
Serial Scan	SS*n*	In serial, scan an input short run of size $n \in [2\text{-}32]$ into a prefix-sum output run. The scan output can either be inclusive or exclusive. ~Two I/Os per element (one on input, one on output.)	

Table 6.2: Basic nomenclature and symbols for the Reduce and Scan primitives.

CPU Serial Implementations: The serial implementations of the Reduce and Scan primitives on a von Neumann CPU are similar (see Table 6.3).

Serial Reduce	**Serial Scan (*Inclusive*)**	**Serial Scan (*Exclusive*)**
 Initial Sum Data ... Final Sum 𝕀 Work: $O(n) = n$ Depth: $O(n) = n$	 Input: ... Inclusive Scan Output: ... Total Sum	 Input: ... Identity Exclusive Scan Total Sum Output: ...

`Reduce( sum, A, n, ⊕, 𝕀 )`
`sum = 𝕀;  // Identity`
`// Reduce`
`for i in 1..n`
`  sum = sum ⊕ A[i];`
`end for`
	`Scan_Inclusive( S,A,⊕,𝕀 )`
	`sum = 𝕀;  // Identity`
	`// Inclusive Scan`
	`for (i = 0; i < n; ++i)`
	`  sum  = sum⊕A[i];`
	`  S[i] = sum;`
	`end for`
		`Scan_Exclusive( S,A,⊕,𝕀 )`
		`sum = 𝕀;  // Identity`
		`// Exclusive Scan`
		`for (i = 0; i < n; ++i)`
		`  S[i] = sum;`
		`  sum  = sum⊕A[i];`
		`end for`

Input: +, 0, [1 2 3 4 5 6 7 8] [0+1+2+3+…+8] **Output:** [36]	**Input:** +, 0, [1 2 3 4 5 6 7 8] [1, 1+2, 1+2+3, ⋯, 1+2+…+8] **Output:** [1 3 6 10 15 21 28 36]	**Input:** +, 0, [1 2 3 4 5 6 7 8] [0, 0+1, 0+1+2, ⋯, 0+1+2+…+7] **Output:** [0 1 3 6 10 15 21 28] ~~36~~

Table 6.3: *Serial Reduce* and *Inclusive & Exclusive Serial Scan.* All three procedures initialize a running sum to identity, then traverse the input array and accumulate new values. Reduce returns the final sum as its output. Both versions of scan output the current running sum for each input. The *exclusive* scan is effectively shifted over one element to the left of the *inclusive* scan. The top, middle, and bottom rows give symbolic depictions, pseudo-code, and examples for each of the three operations.

As shown in Table 6.3, Both Reduce and Scan initialize an accumulator to identity and then, as the code sequentially traverses the data, accumulate the running sum. Reduce writes out the final sum only; whereas Scan writes out the current running sum (inclusive or exclusive) for each input element. The exclusive scan is easy to obtain from the inclusive scan by prepending the identity, reaching back one entry, and dropping the total sum, or final value.

GPU Parallel Implementations: As will be seen in this case study, GPU parallel implementations to achieve high throughput are more complex than CPU serial implementations. My main performance goal was to achieve a solid percentage of peak throughput for both primitives. I achieved solid throughput performance for Reduce and Scan in four main ways:

1) I support TLP via multiple thread warps ‹*nWarps*› per thread block, respecting constraints on *occupancy*.

2) I support ILP via multiple work items ‹*nWork*› per thread.
3) I support coalescence, transferring 32 data elements in a single I/O instruction, for data in main memory.
4) I mitigate bank conflicts for transfers between shared memory and registers.

Throughput Results: Since this chapter is quite long, I give the initial throughput results here to whet your appetite for the details of the rest of this chapter. The main takeaway of this chapter is that my Row DASk provides an excellent starting point for implementing these Reduce and Scan primitives. Of course since the Reduce and Scan primitives are more complex than the simple Copy primitive from Chapters 4 and 5, I also had to overcome some performance hindering issues and use some cleverness in my implementations. The results speak for themselves. Figure 6.4 shows that the Reduce and Scan primitives can achieve nearly the same peak throughput of the simple Copy primitive.

Throughput	**GTX 580** (*Fermi*)			**GTX Titan** (*Kepler*)		
	Reduce	***Scan***	***Copy***	***Reduce***	***Scan***	***Copy***
***Baseline* (GB/s)**	24.2	33.6	49.4	40.0	53.7	86.0
***Best* (GB/s)**	172.7	164.8	175.0	227.0	225.0	236.3
Speedup	**7.1×**	**4.9×**	**4.4×**	**5.7×**	**4.2×**	**2.75×**

Table 6.4: Best throughputs (in gigabytes per second) for Reduce and Scan on the GTX 580 and GTX Titan respectively, and speedups over baseline throughputs. Copy throughputs using the Grid DASk from Chapter 4 are also included for comparison.

The performance results for Reduce and Scan also show that choosing the right ILP and TLP parameters based on extensive experiments results in much better throughput than naively implementing the sequential algorithm on the GPU. For example, the Reduce primitive is up to 7.1× faster than the baseline on the Fermi architecture (GTX 580) and the Scan primitive is up to 4.2× faster than the baseline on the Kepler architecture (GTX Titan)).

Moreover, the four plots in Figure 6.5 show how TLP, ILP, and two different approaches to handling bank conflicts (mitigate or avoid) all contribute to improving throughput. Each plot contains five curves, described briefly in the next paragraph.

The easy way to experiment with ILP is to vary the number of work items per thread (*nWork*), in the range 1-8. The easy way to experiment with TLP is to vary the number of thread warps per block (nWarps), also in the range 1-8. Varying these (and other data access parameters, described later) gives the five curves in each plot: The *Baseline* ‹*nWarps*=1, *nWork*=1 › curve has no extra ILP or TLP; The *ILP-Focused* ‹1, varies› curve increases the work per thread; The *TLP-Focused* ‹varies, 1› increases the warps per thread block. The *Mitigates Bank Conflicts* curve increases both ILP and TLP and allows bank conflicts. However, it uses simple code, which allows CUDA to mitigate the impact of serialized replays. The *Avoid Bank Conflicts* curve, on the other hand, also increases ILP and TLP, but it uses complex code, which allows it to completely avoid bank conflicts. As can be seen from the plots in Figure 6.1, increasing both ILP and TLP achieves the best throughput.

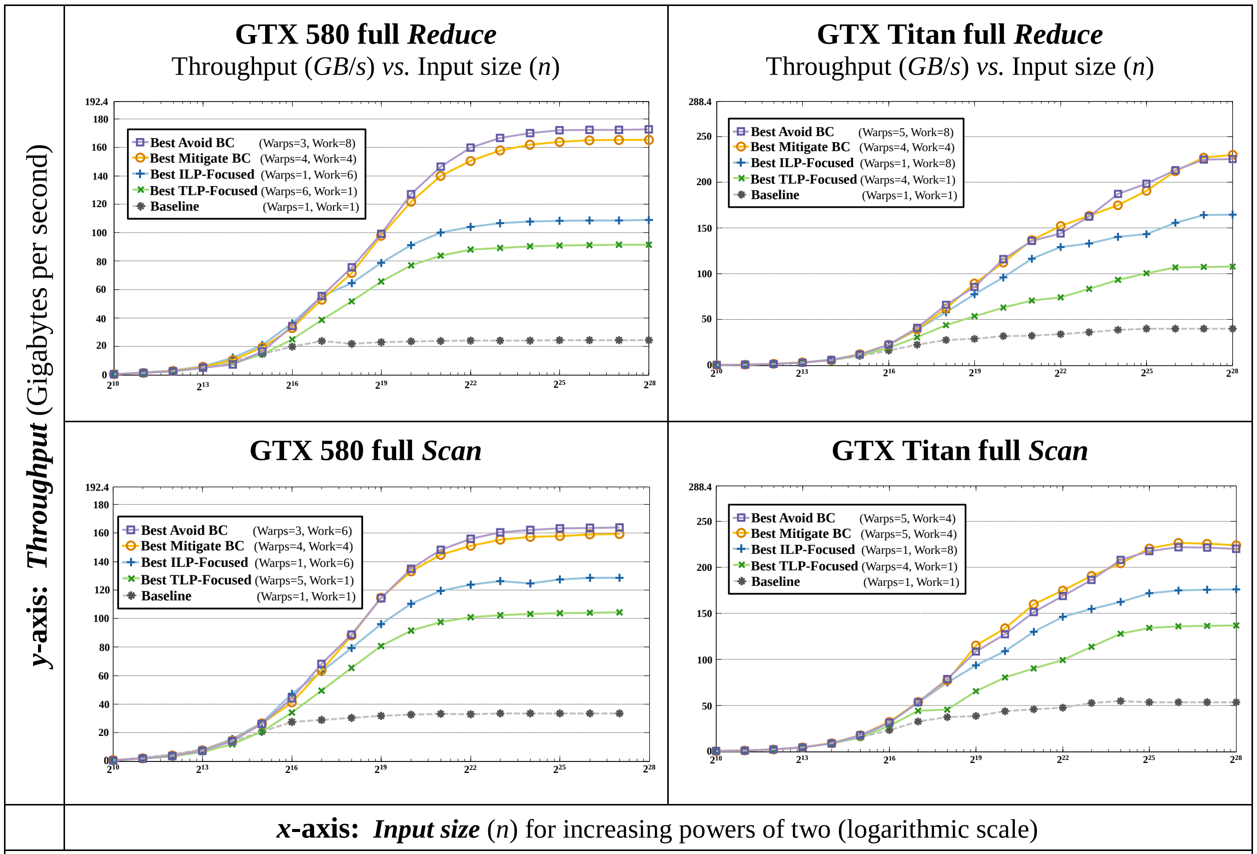


Figure 6.1: Full Reduce and full Scan throughput results (*y*-axis: gigabytes per second (GB/s)) as a function of input size (*x*-axis; log scale) on the GTX 580 {Fermi architecture} (left column) and GTX Titan {Kepler architecture} (right-column) GPUs.

6.2 Issues affecting GPU Performance of Reduce and Scan

For this case study, I measure GPU performance using two metrics – I/O throughput (in giga-bytes per second) and Total Cycles[3] (TC).

Adapting algorithms to the GPU is a challenge because of the many choices for thread organization, register assignment, memory layout, and synchronization. The three main GPU issues for this case study are converting between conflicting memory views, mitigating bank conflicts, and handling constraints on occupancy.

Converting between conflicting memory views: My decision to use associativity but to forbid commutativity in my implementations requires working with short runs of consecutive sequences. However, for better throughput, I must access data in global memory in a way that respects coalescence--recall from Chapter 3.4, that for data that is aligned, coherent, and fully used, coalescence means that the 32 threads in a thread warp can read or write up to 32 data values at the cost of a single I/O.

Consider two examples of global-memory access patterns: a warp-strided access pattern, in which each thread in a warp accesses a unique element in the data warp and then strides to the next data warp (*stride* = *WarpSize* = 32), can run at near peak I/O throughput. In contrast, A sequential, per-thread access pattern (*stride* = 1) can decrease throughput by up to 32×. As we will see later in this case study, to improve performance, I use a coalesced view of data to load from global memory, but convert in shared memory to a sequential view of data that can be placed in registers.

Bank conflicts: A *k-way* bank conflict occurs when *k* threads within the same thread warp access the same bank of shared memory at the same time. The GPU hardware serializes access requests as *k* replays from conflicting threads to ensure correct sequential behavior and in so doing decreases throughput *k*-fold.

[3] Recall from Chapter 3 that *total cycles* (TC) can be computed from two GPU hardware counters as TC=II/IPC, where II = total *instructions issued*, and IPC = average *instructions* retired *per cycle*.

Constraints on occupancy: Recall from Chapter 3.3, that I consider NVidia's occupancy (*ActiveWarps* / *MaxWarps*) metric as a rough measure of the potential for hiding latency using TLP. In general, kernels with higher occupancy tend to have better performance.

Occupancy is initially limited by the programmer's choice of the CTA parameters (thread block & grid sizes), and register and shared memory usage within a kernel may further limit occupancy. For best throughput, GPU programmers should also balance the CTA grid size evenly across the SMs.

6.3 Related Work

The Reduce and Scan primitives come to GPUs originally from the computer hardware community. For small input sequences (or runs*)*, Reduce and Scan can be implemented via the same techniques used by hardware adders for fast binary addition, with the hardware adders becoming software prefix sums.

Hardware Adders: Charles Babbage (Ladner and Fischer, 1980) developed the first linear adders (ripple-carry and carry-add) for his mechanical difference engine. Modern CPU architects later created hierarchical prefix adders that improved performance from linear $O(n)$ to logarithmic $O(\log n)$. David Harris (Harris and Sutherland, 2003) grouped hardware prefix adders into three categories—Sklansky, Kogge-Stone, and Brent-Krung (Sklansky, 1960; Kogge and Stone, 1973; Hillis and Steele, 1986; Brent and Krung, 1982 respectively) —with three hybrid categories—Ladner-Fischer, Han-Carlson, and Knowles[4] (Ladner and Fischer, 1980; Han and Carlson, 1987; and Knowles, 1999 respectively). The Sklansky, Kogge-Stone, and Brent-Krung adders are also known in the software community as "Upper/Lower", "Doubling" and "Even/Odd" prefix-sums respectively.

[4] Knowles style adders use special *kill-bit* hardware and in general cannot be implemented directly in software.

Table 6.5 summarizes the information about each adder type. According to Harris's taxonomy (Harris and Sutherland, 2003), adders are differentiated in terms of three numbers <*l*, *f*, *t*>, where *l* = *Logic levels* (log *n* + *l*), *f* = *Fanout* ($2^f + 1$) and *t* = *Wire Tracks* (2^t). Most hardware adders consist of bit-adder nodes (add two bits and a carry), wires that carry the binary inputs into the nodes, and buffers that store temporary results. In general, parallel adders with low depth tend to be fastest. Adders with low work (nodes) tend to use less silicon and power.

Hardware Name	**Other Names**	**Fanout**	**Tracks**	**Work:** (#Nodes)	**Depth:** (#Logic Levels)
Ripple-Carry	Linear	1	1	= *n*-1	= *n*-1
Sklansky	Upper/Lower	n/2	1	= (*n*/2) log *n*	= log *n*
Kogge-Stone	Doubling	2	n/2	≤ *n* log *n*	= log *n*
Brent-Kung		2	1	≤ 2*n*	= (2 log *n*) - 1
Ladner-Fischer		n/4	1	≤ 4*n*	= log *n* + 1
Han-Carlson		2	n/4	≤ (*n*/2) log *n*	= log *n* + 1
Knowles		varies	1	≤ *n* log *n*	= log *n*

Table 6.5: *Adder Summary* - Area & power is proportional to work. Parallel performance is proportional to depth. Blue is best in category (asymptotically, for Work & Depth).

Figure 6.2 presents a graphical layout of each adder type for a run of length 16. As will be seen later in the case study, for short sequential scans (*Serial-Scans*), the Sklansky layout performs best on short runs (($n \leq 16$) and the Brent-Kung layout performs best for longer runs ($n > 16$). For parallel scans across all the threads in warp (*Warp-Scans*), the Kogge-Stone layout performs best for short runs ($n \leq 32$) and a nested hybrid scan method work best for longer runs ($n > 32$).

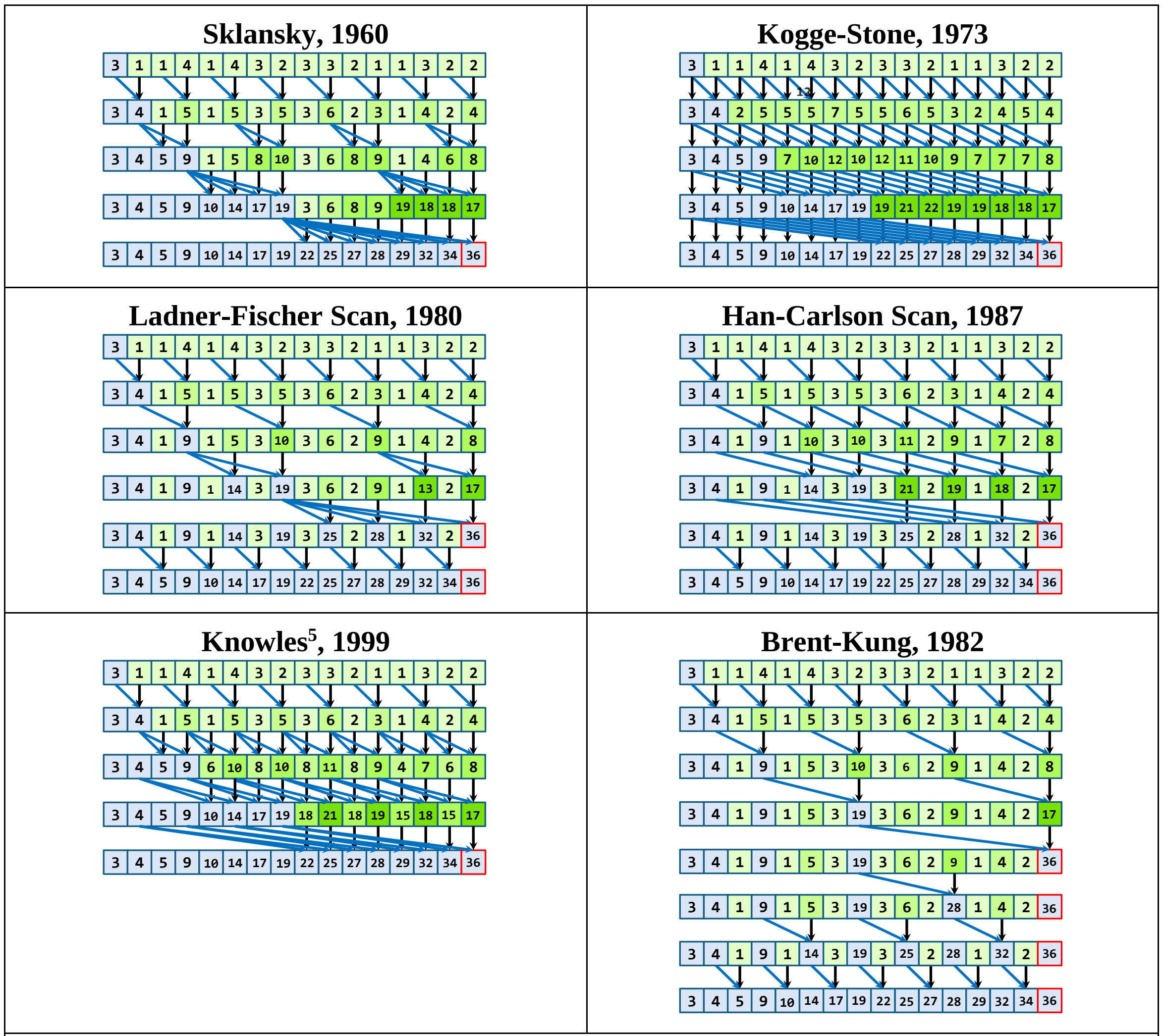


Figure 6.2: Hardware adder diagrams that can be adapted into software reduce and scan (prefix sum) methods. Blue (reach back) and Black arrow pairs represent sum operations. Shades of green represent different levels of partial completion. Blue-Grey boxes represent final results with red outlines containing the total sum.

[5] Knowles adders represent an entire family of layouts of which this is just one instance. This particular layout happens to work correctly as a prefix sum in software, most Knowles layouts do not support prefix sums in software due to the presence of special kill-bit hardware.

GPU Reduce and Scan: Previous work on GPUs include work by Mark Harris (Harris, 2007), who took a naïve GPU reduce for large data sets, and iteratively improved it through a series of eight rewrites, increasing throughput 30×. Though Dr. Harris's final solution is elegant, it is not generic (hardcoded for unsigned integers) and requires commutative summations because he reorders the data layout. Dr. Harris (Harris et al, 2008) later adapted Blelloch's *prefix-sum* (Blelloch, 1990) for a GPU Scan primitive that, despite being both depth-efficient and work-efficient, maps poorly onto GPU hardware and therefore produces poor throughput.

Duane Merrill (Merrill and Grimshaw, 2010, Parallel Scan) used high-level parallel patterns (Reduce-then-Reduce; Scan-then-Fan; and Reduce-then-Scan) and an improved low-level Warp-Scan based on the Hillis-Steele variant of the Kogge-Stone prefix sum ((Hillis and Steele, 1986; and Kogge and Stone, 1973 respectively) to produce better mappings of Reduce and Scan onto GPUs. Several other of Dr. Merrill's clever ideas include storing arrays in registers, scanning data blocks using a 3-level local hierarchy, accessing memory using efficient I/O skeletons, and inlining massive amounts of code using C++ templates. I use many of Dr. Merrill's ideas in my own implementations.

6.4 Parallel Patterns

Improvements in Reduce and Scan throughput are also obtained by combining common parallel patterns for each primitive.

6.4.1 Reduce Parallel Patterns

For the Reduce primitive, the singleton output sum depends on all input values. Figure 6.3 illustrates two natural parallel patterns to reduce *n* elements using *p* threads: Tree Reduce and Run Reduce (AKA Reduce then Reduce) (Merrill and Grimshaw, 2010, Parallel Scan). I use both patterns at different levels of my nested Reduce GPU implementation.

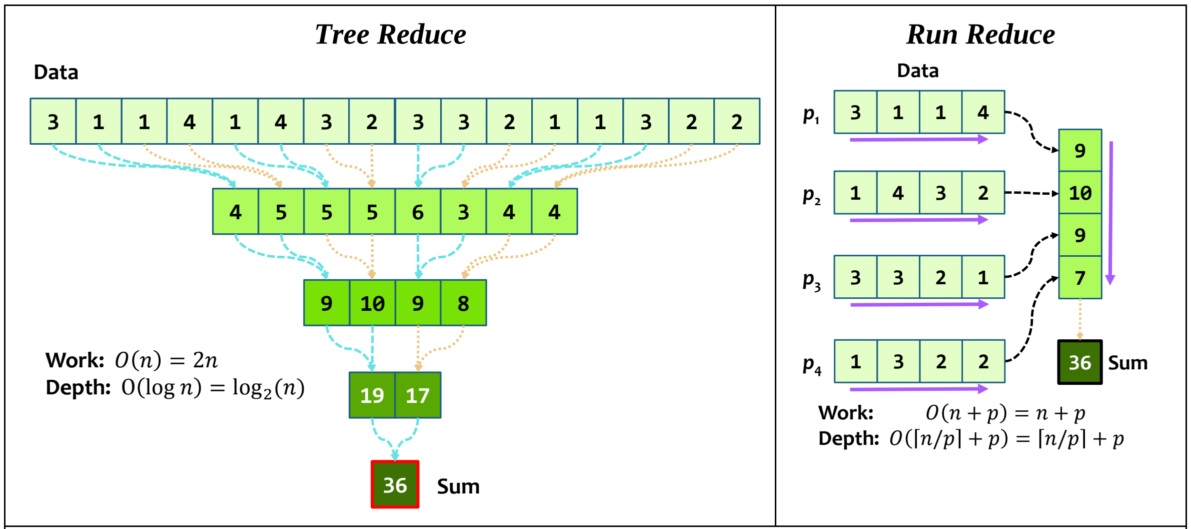


Figure 6.3: *Parallel Reduce Patterns.* This figure Illustrates two parallel patterns to reduce n elements using p threads. In the left panel is a *tree reduce* pattern of 16 elements in 4 = $\log_2(16)$ stages. In the right panel is a *Run Reduce* pattern of 16 elements in two stages (serially reduce 4 element runs into run sums using 4 threads, *serially reduce* 4 run sums into a final sum using 1 thread).

The Tree Reduce pattern (see Figure 6.3, left panel), a *fine*-grained reduction, is easy to visualize and implement. It reduces two input elements per thread to one output sum at each stage and, in so doing, takes $\log_2(n)$ stages in total to fully reduce n elements down to one final sum. Total work is linear = $2n$ = $n+n/2+n/4+\ldots+2+1$. Total depth is logarithmic = $\log_2(n)$. Total I/Os is linear = $3n$ (or $2n$).

Tree Reduce, however, has three distinct disadvantages:

- Its relatively high kernel launch costs, since $\log_2(n)$ stages are required
- Its suboptimal processor utilization, since each stage launches half the threads of the previous stage
- Its relatively high I/O transfer costs, since intermediate sums from each stage must be transferred to the next using $3n$ I/Os if all sums are stored and $2n$ I/Os if continuing threads keep their sums

The Run Reduce pattern (see Figure 6.8 right panel), a *coarse*-grained reduction, is even easier to visualize and implement. It partitions the n data elements across p threads (cores) into even runs (*run length*=$\lceil n/p \rceil$). In the first reduce stage, each thread serially reduces its assigned run, producing p run-sums (one per run). In the second reduce stage, the p run-sums are serially reduced by a single thread (core) to a final sum. Total work is linear = $n+p$. Total depth is linear = $\lceil n/p \rceil + p$. Total I/Os is linear = $n+p+1$. Choosing $p = \sqrt{n}$, results in minimum depth = $2\sqrt{n}$ and work = $n+\sqrt{n}$.

As described, the second stage of Run Reduce has essentially the worst possible processor utilization: only one thread is active. As a result, in my actual GPU implementation, I replace the second stage's serial reduce by a nested run-reduce and tree-reduce to involve more parallel threads.

6.4.2 Scan Parallel Patterns

Like Reduce, Scan looks at all data values. Unlike Reduce, Scan must output a prefix sum for each input data value that depends either on all previous data values or on some previous data values or sums. There are two natural Scan patterns (see Figure 6.4), Scan-then-Fan and Reduce-then-Scan. With both Scan patterns, the input data is partitioned evenly into per-thread runs. For my own GPU Scan implementation, I use both patterns at different levels of the CTA hierarchy.

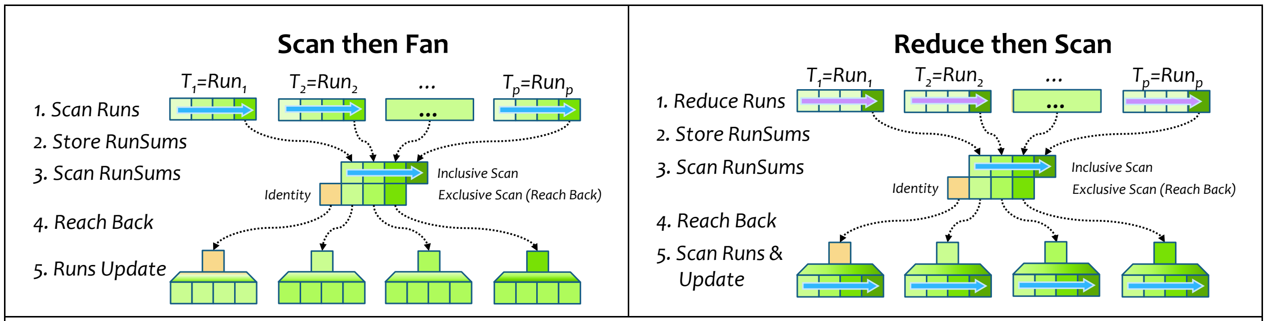


Figure 6.4: Two Parallel Scan patterns 1) *Scan-then-Fan* (Top panel), and 2) *Reduce-then-Scan* (Bottom panel).

***Scan-then-Fan*:** Scan-then-Fan (see Figure 6.4, left panel) is similar to the Run-Reduce pattern and takes five stages:

1. **Scan Run:** Each thread serially scans its assigned run. Each run is locally correct but is missing a prefix sum from all preceding thread runs.
2. **Store Run-Sums:** Per-thread run-sums from stage 1 are stored in another array.
3. **Scan Run-Sums:** The per-thread run-sums from stage 1 are inclusively scanned.

4. **Reach Back:** The missing prefix-sums needed in stage 1 are found as the exclusive scan of the stage 1 run-sums, which were obtained from the inclusive scan by reaching back one entry.
5. **Run Update (AKA fan):** The missing prefix-sum for each thread is accumulated into all locally scanned elements in each per-thread run to generate the final scanned results. This stage is also known as a *fan*.

The Scan-then-Fan pattern takes ~ 4 I/Os per data element, since both stage 1 and stage 5 must read and write each element. The Scan-then-Fan pattern with input size *n* and *p* threads uses linear *Work* = 2*n*+2*p*, linear *Depth* = 2[*n*/*p*]+2*p*, and linear *Total I/Os* = 4*n*+4*p*.

***Reduce-then-Scan*:** The Reduce-then-Scan pattern (Figure 6.9, right panel) is derived from the Scan-then-Fan pattern. In stage 1, Scan Run, the serial Scan is replaced by a serial Reduce to compute the stage 1 run-sums, and in stage 5, Fan, the run update becomes a serial Scan initialized with the missing prefix-sums from stage 4. This scan pattern decreases the total I/Os per element since the first stage now writes a single run sum instead of an entire run. The Reduce-then-Scan pattern with input size *n* and *p* threads uses linear *Work* = 2*n*+2*p*, linear *Depth* = 2[*n*/*p*]+2*p*, and *Total I/Os* = 3*n*+4*p*.

6.5 Reduce and Scan Overview

For correctness and for good GPU throughput, both of the Reduce and Scan primitives need to do the following:

- Map their parallel patterns onto the GPU's 2-level CTA hierarchy
- Support consecutive access for non-commutative summation
- Support both ILP & TLP
- Mitigate GPU issues of converting between warp and sequential views, bank conflicts, and constraints on occupancy

At a high level, My GPU Reduce primitive is based on the Run-Reduce pattern and is implemented using two GPU kernels — `GPU_Reduce` and `GPU_SumRows`. My GPU Scan primitive is based on the Reduce-then-Scan pattern and is implemented using three GPU kernels — `GPU_Reduce`, `GPU_SumsToStarts`, and `GPU_Scan`. The `GPU_Reduce` and `GPU_Scan` kernels do most of the work for each primitive.

The basic idea behind these two implementations, as with so many algorithms, is to divide the large Reduce and Scan problems into smaller instances and apply recursion. This division, however, is complicated by the need to work with the GPUs 2-level CTA hierarchy (grid of thread blocks, threads within a thread block). To support sequential access within the 2-level CTA, I implement the following two-part solution:

> **CTA Level 1 (grid of thread blocks):** For consecutive data access at the first CTA level, both `GPU_Reduce` and `GPU_Scan` use my Row DASk, described in Chapter 5. This DASk partitions data into fixed-size data blocks, which it then distributes across a fixed number *r* of rows, resulting in *c* columns. It then assigns each of the *r* data rows to a thread block in the grid. Each thread block subsequently works along its assigned data row sequentially, data block by block.
>
> **CTA Level 2 (threads within a thread block):** Consecutive access within the data block by each thread in the thread block is handled by the `BlockReduce` and `BlockScan` methods. Both methods take as input a full fixed-size data block. The 3-level nested `BlockReduce` method reduces the entire data block to a single block-sum, which is then accumulated into the current row-sum. The 3-level nested `BlockScan` method sequentially[6] scans the entire input block into a scanned prefix sum (inclusive or exclusive as requested), which is updated with the current row-start (missing row prefix). The block of scanned results is then output.

Reduce Overview: With the `GPU_Reduce` kernel, each thread block initializes a row-sum to identity and then marches along its assigned data row, block by block, calling `BlockReduce` on each data block.

[6] I would have liked to have skeletonized these methods as *sequential block access skeletons (*BASks), but because both methods depend on pass-through parameters from my Row DASk as well as parameters for ILP, TLP, occupancy, and bank conflict mitigation, both methods are too unwieldy to generalize.

After each `BlockReduce` call, each thread block accumulates the resulting block-sum into the current row-sum. The `GPU_Reduce` kernel generates *r* row-sums as output that still need to be reduced to the final *total-sum*.

Since the `GPU_Reduce` kernel outputs *r* row-sums to be consumed as input by the `GPU_SumRows` kernel, I deliberately choose my fixed number of rows (*r*) to be small ($r \leq 1{,}000$). This allows `GPU_SumRows` to reduce the *r* row-sums to the final sum as output using a single instance of the `BlockReduce` method with a single thread block (*GridSize* =1). Calling the `GPU_Reduce` kernel followed by the matching `GPU_SumRows` kernel implements the full GPU Reduce primitive.

Efficient I/O access for Reduce: Given *r* rows and *n* input values and assuming *r* is much less than *n* ($r \ll n$),, then the Reduce primitive needs only a little over one global memory transfer per data warp on average. The `GPU_Reduce` kernel reads each input exactly once from global memory to reduce *n* inputs to *r* row-sums `GPU_SumsRows` reduces those *r* row sums into the final sum. Coalescence is respected by using the warp-by-warp BASk to load input. This arrangement means that I only need one transfer per data warp (32 data elements), which results in $\left\lceil\frac{n}{32}\right\rceil$ total data transfers. Combining transfers from both kernels results in $\left\lceil\frac{n}{32}\right\rceil + r + \left\lceil\frac{r}{32}\right\rceil + 1 = O(n + r)$ total I/Os for the GPU Reduce primitive.

Scan Overview: For the GPU Scan primitive, I must generate the missing row prefixes for each data row before locally scanning each data block along each data row. The `GPU_Reduce` kernel generates *r* row-sums. The `GPU_SumsToStarts` kernel exclusively scans the *r* row-sums as input into *r* row-starts as output using a single instance of the `BlockScan` method with a single thread block (*GridSize* = 1). These *r* row-starts provide the missing row prefixes for globally correct scan results along each data row. Finally, with the `GPU_Scan` kernel each thread block loads its missing row-start (row prefix) from the row-starts array and then marches along its assigned data row, block by block, locally scanning each data block by calling `BlockScan`. The current row-start is also accumulated as a prefix into the local scanned results to generate global scanned results that are then output. After each data block is scanned, the resulting block-sum is accumulated into the current row-start to prepare for the next data block along the row. Calling all three scan kernels in order (`GPU_Reduce`, `GPU_SumsToStarts`, and `GPU_Scan`) implements the full GPU Scan primitive.

Efficient I/O access for Scan: Given r rows and n input values and assuming r is much less than n ($r \ll n$), then the GPU Scan primitive needs a little over three global memory transfers per data warp on average. `GPU_Reduce` reads each input exactly once to reduce n inputs to r row-sums; `GPU_SumsToStarts` exclusively scans these r row-sums into r row-starts; and `GPU_Scan` reads each row-start. `GPU_Scan` then reads a data block, combines the row-start with the scan of the data block, and writes out the final prefix sum. I respect coalescence in my `GPU_Reduce` and `GPU_Scan` kernels by loading input and storing output using a warp-by-warp view of global memory. This approach means that I only need one transfer per data warp (32 data elements) and results in $\left\lceil \frac{3n}{32} \right\rceil$ total data transfers across both kernels (`GPU_Reduce` and `GPU_Scan`). Combining transfers from all three kernels results in $\left\lceil \frac{3n}{32} \right\rceil + 2r + \left\lceil \frac{2r}{32} \right\rceil = O(n + r)$ total I/Os for the GPU Scan primitive.

6.6 Reduce and Scan Implementation Details

To implement Reduce and Scan, I present several helper methods in a bottom-up manner.

- 6.6.1 *Run Load*/*Store*: Each thread transfers a short run of data between memory and registers.
- 6.6.2 *Serial Reduce*/*Serial Scan*: Each thread serially reduces or scans a short run of data kept in registers.
- 6.6.3 *Warp Reduce*/*Warp Scan*: All threads in one thread warp cooperatively scan a single data warp kept in shared memory. The final-sum is found in the last column after the scan completes.
- 6.6.4 *Run Update*: Each thread sums a common prefix into a short run of data in registers.
- 6.6.5 *Block Reduce*/*Scan*: The *Run Load*/*Store*, *Serial Reduce*/*Scan*, *Warp Reduce*/*Scan*, and *Run Update* methods are nested to recursively reduce or scan an entire data block.

- 6.6.6 Reduce/Scan primitives: The full Reduce primitive on a GPU is implemented using two GPU kernels— `GPU_Reduce`, and `GPU_SumRows`—which follow the Run-Reduce pattern. The full Scan primitive is implemented using three GPU kernels— `GPU_Reduce`, `GPU_SumsToStarts`, and `GPU_Scan`—which together follow the Reduce-then-Scan pattern.

I show a high-level overview of both Reduce and Scan in Figure 6.5. I aim for my Reduce and Scan primitives to be correct, generic, and fast. I verify correctness by comparison against baseline CPU methods. Genericity is supported by C++ template parameters (Alexandrescu, 2001). Throughput experiments on TLP and ILP are supported by two template parameters, ‹*nWarps*, *nWork*›, each in the range {1-8}.

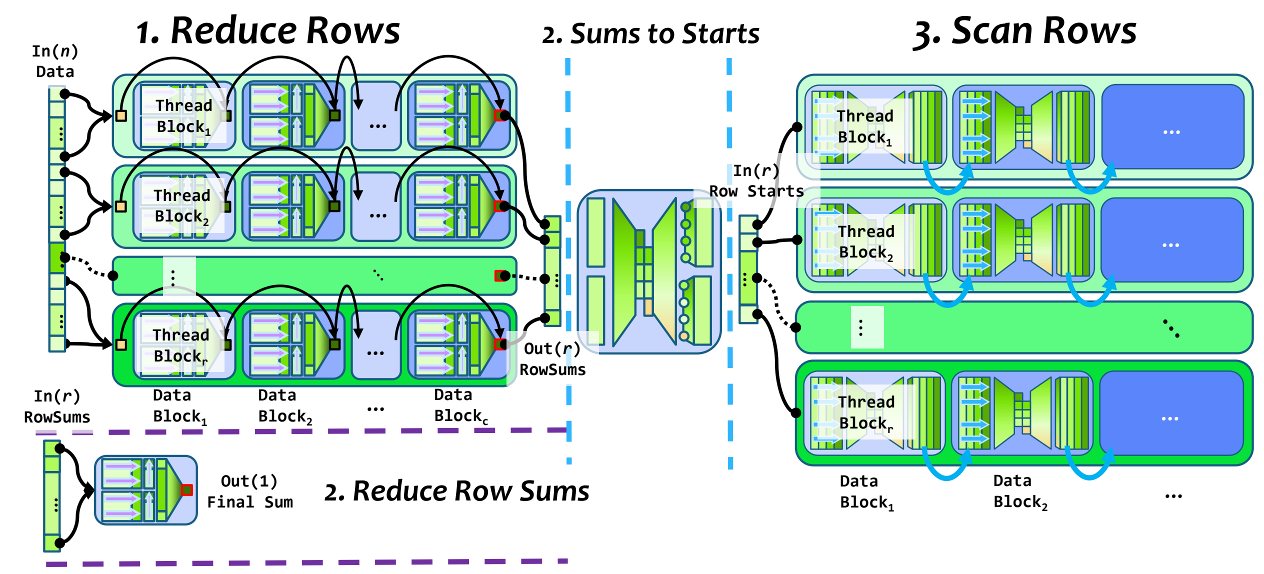


Figure 6.5: High level overview of *Reduce* in two steps (purple lines) and *Scan* in three steps (blue lines). **Reduce:** 1) The reduce rows step reduces *n* data items to *r* row sums. 2) The reduce row sums step reduces *r* row sums to the final sum. **Scan:** 1) The reduce rows step was already discussed. 2) The sums to starts step exclusively scans *r* row sums into *r* row starts. 3) The scan rows step scans *n* data elements into *n* scanned prefix sums using the *r* row starts as starting prefix sums for each row. Both the reduce rows and scan rows steps use my Row DASk to have each thread block march down its assigned data row, processing data in parallel, data block by block. Row sums are carried along each row accumulating block sums after scanning each data block.

6.6.1 RunLoad and RunStore Methods

The RunLoad and RunStore methods transfer short sequential runs between memory (global or shared) and registers. These methods support both sequential and strided access. I briefly discuss these methods in terms of parameters, range checking, total cycles, and issues.

***Parameters*:** Both RunLoad and RunStore require C++ template parameters. The RunLoad method requires three template parameters -- *valT, WorkStride, nWork*. The RunStore method requires four template parameters -- the same three as RunLoad plus *Identity* (𝕀). The *valT* parameter is simply a generic placeholder for different data types. The *nWork* parameter specifies how many data elements each thread loads (stores) per data block, usually {1-8}. Using more elements increases ILP efficiency via software pipelining at the cost of more registers. *WorkStride* specifies the stride between adjacent memory accesses per thread. Consequently, a stride of 1 is sequential access, a stride of 32 supports warp-by-warp access, and a padded stride of 33 (32+1) can help avoid bank conflicts. The *Identity* (𝕀) parameter is used as a default value for out of range loads in *RunLoad*. Since these template parameters are compile time constants, only relevant instructions are compiled into the GPU kernel.

***Range Checking*:** Recall that the Row DASk requires that the programmer implement four body templates: one without range checking **_RC_NONE* (…), and three range-checked versions, **_RC_START*, **_RC_STOP*, **_RC_BOTH*, that check individual loads (stores) against the ranges [*start* …), (… *stop*] and [*start*, *stop*], respectively. Figure 6.6 shows implementations for two of the range intervals, the open (…) and closed [*start*, *stop*] intervals. The implementations for the two half-open intervals [*start*, …) and (…, *stop*] and the family of four RunStore methods are similar.

```
template< valT, WorkStride, nWork >
RunLoad_RC_NONE( S, v1, v2, v3, … )  // *NO* range check
  // Load Run
     // All “if (nWork >= *) { … }” wrapper statements
     // disappear at compile time, leaving just the inner … statement.
     // Any unused statements based on the nWork size also
     // disappear at compile time.
  if (nWork >= 1) { v1 = S[(0*WorkStride)]; }
  if (nWork >= 2) { v2 = S[(1*WorkStride)]; }
  if (nWork >= 3) { v3 = S[(2*WorkStride)]; }
  if (nWork >= 4) { v4 = S[(3*WorkStride)]; }
  ...
end RunLoad
```

```
template< valT, WorkStride, nWork, 𝕀 >  // Range check [start, stop]
RunLoad_RC_BOTH( A, dataOff, start, stop, v1, v2, v3, ... )
  S = (valT *)&(A[dataOff]);     // Get starting pointer
  // Get offsets
  if (nWork >= 1) { GI_1 = dataOff + (0*WorkStride); }
  if (nWork >= 2) { GI_2 = dataOff + (1*WorkStride); }
  if (nWork >= 3) { GI_3 = dataOff + (2*WorkStride); }
  ...
  // Range Check offsets against [start, stop]
  if (nWork >= 1) { T1 = (start <= GI_1) & (GI_1 <= stop); }
  if (nWork >= 2) { T2 = (start <= GI_2) & (GI_2 <= stop); }
  if (nWork >= 3) { T3 = (start <= GI_3) & (GI_3 <= stop); }
  ...
  // Load Run (range check accesses)
  if (nWork >= 1) { v1 = 𝕀; }  // Default to Identity (if out of range)
  if (nWork >= 2) { v2 = 𝕀; }
  if (nWork >= 3) { v3 = 𝕀; }
  ...
  if (nWork >= 1) { if (T1) { v1 = S[(0*WorkStride)]; } }
  if (nWork >= 2) { if (T2) { v2 = S[(1*WorkStride)]; } }
  if (nWork >= 3) { if (T3) { v3 = S[(2*WorkStride)]; } }
  ...
end RunLoad_RC_BOTH
```

Figure 6.6: The `RunLoad_RC_NONE` template method (upper panel) loads a short run from memory into registers without any range checking. The `RunLoad_RC_BOTH` template method (lower panel) is similar but does more work to prevent out of range memory accesses against the range [*start*, *stop*]. Code in light grey gets elided away at compile time.

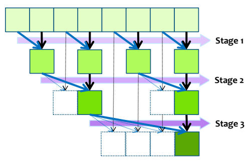
***Total Cycles*:** In this section, I introduce formulas for how many machine cycles these methods take to run. Assume n = run length to load; G & S are the number of cycles to transfer a data value between global memory (G) or shared memory (S) and registers respectively; and k is the number of cycles it takes for an instruction to traverse the instruction pipeline. If so, then the total cycles for *RunLoad* or *RunStore* to transfer a data run is (G + k + n-1) for global memory transfers, and (S + k + n-1) for shared memory transfers.

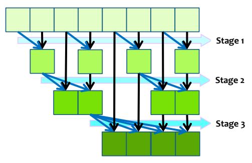
Load/Store Issues: While writing these Load/Store methods, I dealt with two main issues -- first, accessing global memory requires coalescence for good throughput; second, accessing shared memory can lead to bank conflicts. I discuss both issues in more detail in Section 6.7.

6.6.2 `SerialReduce` (SR*n*) and `SerialScan` (SS*n*) Methods

The `SerialReduce` and `SerialScan` methods reduce or scan short data runs of length *n*, one serial run per thread. The short runs are kept in registers for each thread in a thread block (or thread warp). (I denote these methods as SR*n* or SS*n*, meaning SR8 or SS8 for n = 8). Obviously, one could directly apply a sequential reduce or sequential scan on the run kept in registers (as shown in Table 6.3), but such an approach creates RAW dependencies since the output from each sum becomes the input for the next sequential sum and therefore each dependent summation must stall for the length of the GPU instruction pipeline, *k*. Instead, for short runs, I use tree-based adders, which chip architects use to parallelize binary addition, in order to improve ILP for both `SerialReduce` and `SerialScan`. Tree-based adders, such as Sklansky's adder (as shown in Figure 6.2) have dependencies from one tree stage to the next but no dependencies within a stage. Each thread can sum stage by stage and increase ILP by decreasing stalls within stages. A conservative estimate of the total cycles for these serial methods is $nSums + k \cdot nStages$.

For both serial reduce and serial scan, I base my code on the Sklansky adder (see Figure 6.2). The serial reduce (left panel) drops sums that do not contribute to the final sum and takes up to (n-1) + $k \cdot \log_2(n)$ total cycles. This approach is always faster than sequential reduce. The serial scan (right panel) uses all sums in all stages and takes up to $(n/2) \cdot \log_2 n + k \cdot \log_2 n$ total cycles.





Serial Reduce

```
template< valT, sumT, rL >
SerialReduce( runSum, ⊕, v1, v2, … )
// Sklansky Tree-Reduce
if (rL >  2) { v2 = v1⊕v2; } //S1
if (rL >= 4) { v4 = v3⊕v4; }
if (rL >= 6) { v6 = v5⊕v6; }
if (rL >= 8) { v8 = v7⊕v8; }
if (rL >  4) { v4 = v2⊕v4; } //S2
if (rL == 7) { v7 = v6⊕v7; }
if (rL >= 8) { v8 = v6⊕v8; }
// Run Sum (select on run length)
if (rL == 8) { runSum = v4⊕v8; }//S3
if (rL == 7) { runSum = v4⊕v7; }
if (rL == 6) { runSum = v4⊕v6; }
if (rL == 5) { runSum = v4⊕v5; }
if (rL == 4) { runSum = v2⊕v4; }
if (rL == 3) { runSum = v2⊕v3; }
if (rL == 2) { runSum = v1⊕v2; }
if (rL == 1) { runSum = v1; }
end SerialReduce
```

Serial Scan

```
template< valT, sumOp, rL >
SerialScan( ⊕, v1, v2, v3, v4, … )
// Sklansky Scan
  if (rL >= 2) { v2=v1⊕v2; } //S1
  if (rL >= 4) { v4=v3⊕v4; }
  if (rL >= 6) { v6=v5⊕v6; }
  if (rL >= 8) { v8=v7⊕v8; }
  if (rL >= 3) { v3=v2⊕v3; } //S2
  if (rL >= 4) { v4=v2⊕v4; }
  if (rL >= 7) { v7=v6⊕v7; }
  if (rL >= 8) { v8=v6⊕v8; }
  if (rL >= 5) { v5=v4⊕v5; } //S3
  if (rL >= 6) { v6=v4⊕v6; }
  if (rL >= 7) { v7=v4⊕v7; }
  if (rL >= 8) { v8=v4⊕v8; }
end SerialScan
```

Figure 6.7: *Serial Reduce* and *Serial Scan* for an 8 element run [*v1-v8*] in 3 stages ($\log_2 8$). Code in light grey gets elided away at compile time.

For serial scan, I tested the Sklansky, Kogge-Stone, and Brent-Kung adders (see Figure 6.8), and found that Sklansky is fastest for scans on short runs ($n \leq 16$) and for $n \leq 64$ it is faster than sequential scan. Brent-Kung is fastest on longer runs ($n > 16$) and will always be faster than a sequential scan. For both serial reduce and serial scan the number of registers grows linearly in n, so I recommend keeping sequential runs short ($n \leq 8$) to avoid undue register pressure. Being pragmatic, this means that there is no need for the Brent-Kung version of serial scan on current GPU architectures.

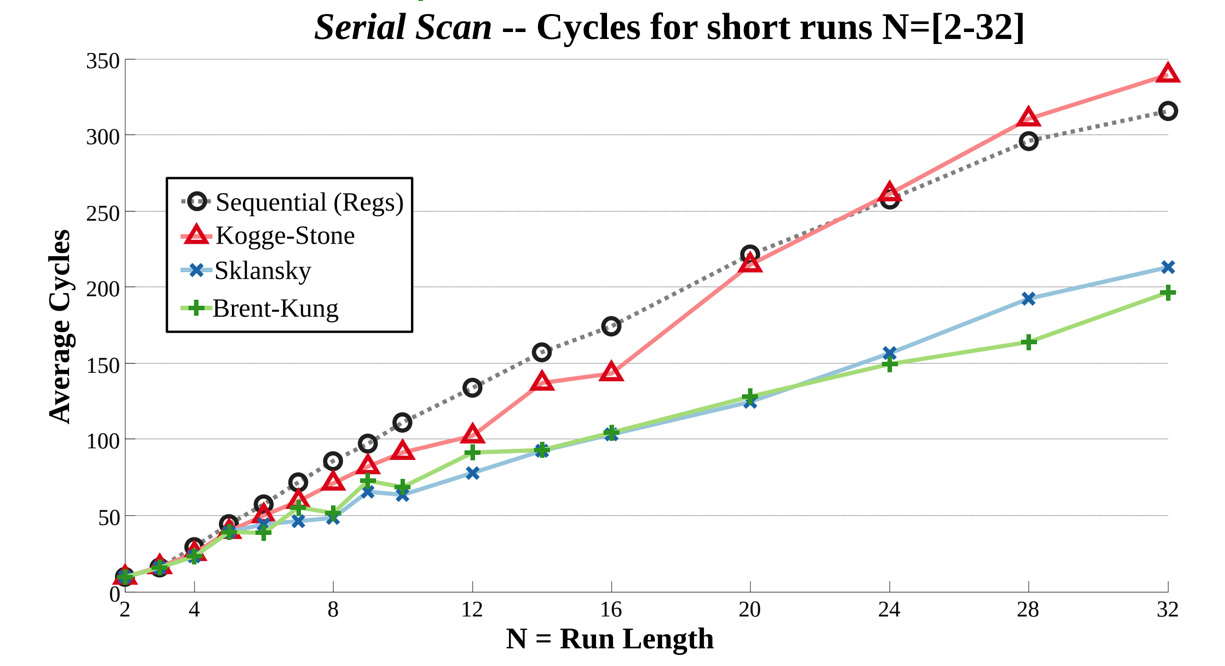


Figure 6.8: *Serial Scan Total cycles* – Total cycles usage by *Sequential* (in regs), *Kogge-Stone*, *Sklansky*, and *Brent-Kung* methods for increasing run lengths, *n* in [2..32]. Note: Not shown Sequential (in shared memory) was about 5.3× slower (1684.4/315.4) than *Sequential* (in regs) in the worst case (*n*=32). For *n* = 8, Sklansky is about 9.2× faster [445.5/48.3] than Sequential (in shared memory) and about 1.77× faster [85.7/48.3] than Sequential (in regs).

Parameters: `SerialReduce` and `SerialScan` both take three template parameters -- *valT*, *sumT*, and *rL*. The *valT* parameter provides a generic place holder for different data types to scan. The *sumT* parameter is the data type of the sum operator (typically as a functor). The *rL* parameter is the run length of the data run in registers to be scanned. These methods also have function parameters, up to eight input register values [*v*1..*v*8] to scan (all of type *valT*), a sum operator ($\oplus$) of type *sumT*, and either outputs the *runSum* for serial reduce or outputs the inclusive scan in those same 8 registers [*v*1..*v*8].

Total Cycles: Assume *n* is the length of the short run to scan and *k* is the total number of cycles it takes for an instruction to traverse the instruction pipeline. If so , the total cycles (TC) these Sklansky tree-based `SerialReduce` and `SerialScan` methods take are TC $\leq (n\text{-}1) + k\cdot\log_2 n$ and TC $\leq (n/2)\cdot\log_2 n + k\cdot\log_2 n$ respectively. For short runs, these results compare favorably with the more obviousSequential Serial Reduce and Scan, which both take TC = $k(n\text{-}1)$ cycles. On the GTX Titan (Kepler) for a run length of 8, my tree-based serial scan is 9.2× faster than a sequential scan in shared memory (48.3 vs. 446.0 total cycles) and 1.77× faster than a sequential scan in registers (48.3 vs. 85.7 average total cycles).

As written, my tree-based `SerialReduce` and `SerialScan` code should take exactly *n* registers and emit exactly *n*-1 and (*n*/2)·$\log_2 n$ sum instructions respectively. Unfortunately, the CUDA compiler, in an attempted optimization, reorders the code, resulting in extra register use and unnecessary stalls that take up to 10% more cycles. I circumvent this by overloading the sum operator to emit .PTX assembly (shown in Figure 6.9).

```
// Generic ADD functors
template < typename valT >
struct CPU_Add
{
  __host__ __forceinline__
  valT operator()( valT a, valT b )
    {
        return (a+b);
    }
};
template < typename valT >
struct GPU_Add
{
  __device__ __forceinline__
  valT operator()( valT a, valT b )
    {
        return (a+b);
    }
};
```

```
// Overload via Partial Specialization
template <>
struct GPU_Add< U32 >
{
  __device__ __forceinline__
  U32 operator()( U32 a, U32 b )
    {
      U32 c;
      asm volatile (
        "add.u32 %0,%1,%2;\r\n"
        : "=r"(c) : "r"(a), "r"(b) );
      return c;
    }
};
template <>
struct GPU_Add< U64 >
{
  __device__ __forceinline__
  U64 operator()( U64 a, U64 b )
    {
      U32 c;
      asm volatile (
        "add.u64 %0,%1,%2;\r\n"
        : "=l"(c) : "l"(a), "l"(b) );
      return c;
    }
};
```

Figure 6.9: Add functors for CPU & GPU (left panel) and overloaded ADD functors for the GPU (right panel) that emit .PTX assembly to mitigate the de-optimization issue. The `volatile` keyword in the `asm` context requests that the CUDA compiler should avoid optimizing the wrapped .PTX assembly instructions.

(The `volatile` modifier on the `asm` keyword requests that the CUDA compiler should avoid optimizing the wrapped .PTX assembly instructions, which effectively prevents CUDA from reordering the summation chain.) Of course, such an approach means that the CPU and GPU versions of my sum functors need to be separated for correct compilation. All in all, this work-around avoids the unwanted stalls. However, it unfortunately does not prevent CUDA from generating extra registers.

6.6.3 `WarpReduce` (WR*n*) and `WarpScan` (WS*n*) methods

Duane Merrill's [Merrill 2010] efficient `WarpScan` method (denoted as WS*n*, meaning, WS8 for *n* = 8) cooperatively scans an entire data warp using one thread warp (see Figure 6.10).

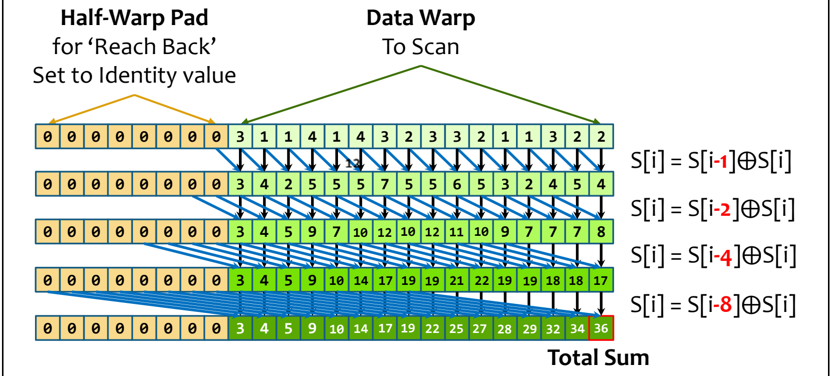


Figure 6.10: Dr. Merrill's *Warp-Scan* based on the Hillis-Steele variant of the Kogge-Stone adder. Orange elements represent identity (𝕀) values.

Dr. Merrill's fixed-size GPU algorithm is based on the Hillis-Steele parallel algorithm (Hillis and Steele, 1986), which is based on the Kogge-Stone adder (Kogge and Stone, 1973). The Hillis-Steele algorithm trades an extra half-warp of pad columns to eliminate branching (leftmost 8 columns of orange identity elements in Figure 6.10). After the scan, the last element in the warp array contains the total sum. Therefore, this `WarpScan` method can also implement `WarpReduce` (denoted as WR*n* ie WR8 for *n* = 8).

For my `WarpScan` and `WarpReduce` experiments, I implemented parallel scan code based on Kogge-Stone, Sklansky and Brent-Kung adders but Kogge-Stone was the clear performance winner (see Figure 6.11). Both Sklansky and Brent-Kung require extra branching operations and more registers. Brent-Kung also requires twice as many logarithmic stages, $2 \cdot \log_2(n) - 1$ vs. $\log_2(n)$.

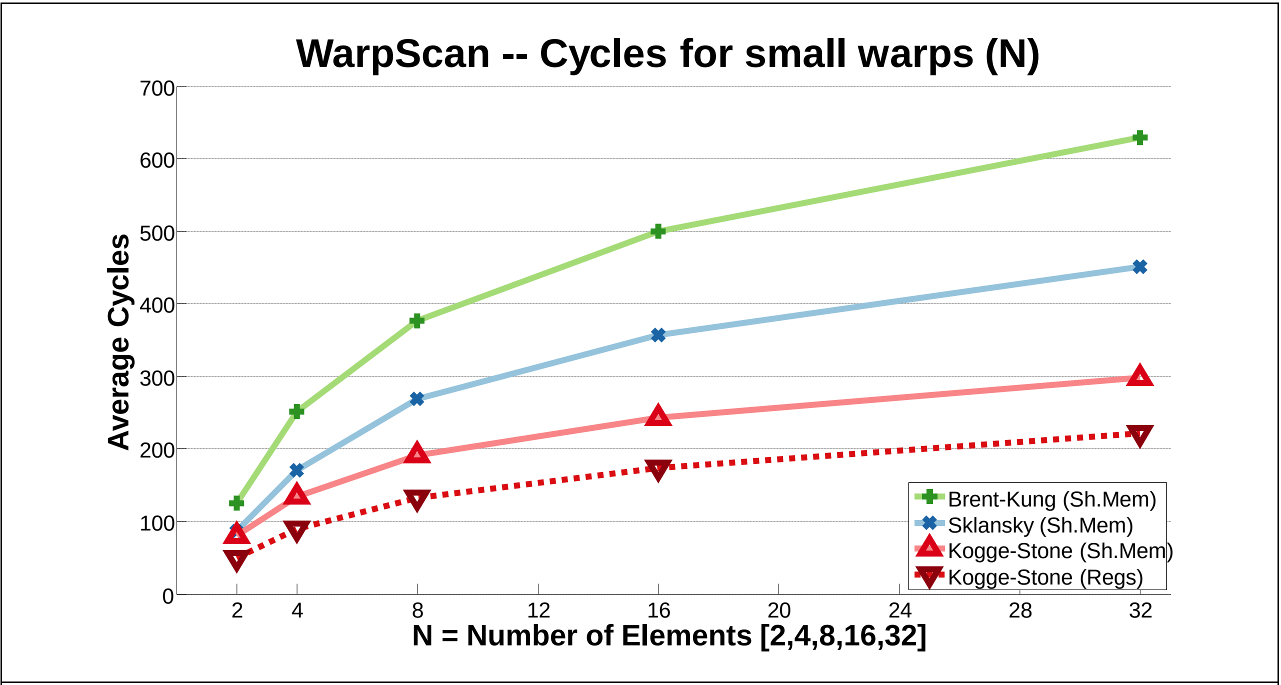


Figure 6.11: ***Warp Scan Total cycles*** – Total cycles used by *Kogge-Stone*, *Sklansky*, and *Brent-Kung* Warp-Scan methods for increasing run lengths, *n* in [2..32] as powers of two on the GTX Titan, All three methods use shared memory to communicate results across threads. Also shown is an alternate *Kogge-Stone* method (dashed line) that uses registers to communicate results across threads via the .PTX SHUFFLE command resulting in fewer cycles.

Figure 6.12 shows two implementations of Warp-Scan based on the Kogge-Stone adder layout. The left panel shows unrolled code that shares intermediate prefix sums across using shared memory. The first iteration reaches back one column (-1), each subsequent iteration doubles the number of columns to reach back. After five iterations ($5 = \log_2(32)$) the exclusive prefix sum for the entire data warp has been computed. The last valid data column contains the total sum (warp-sum) that is needed for Warp-Reduce. This implementation will work for both Fermi and Kepler architectures. The right panel shows .PTX code that shares intermediate prefix sums across threads using registers via the new Kepler specific "Shuffle" command. The "Shuffle" warp-scan is faster and avoids having to use shared memory at a loss of generality as the desired sum operator ($\oplus$) must be hard-coded. `WarpReduce` is the exact same function as `WarpScan`, except the focus is on extracting the total-sum from the last data column in shared memory (or from the last thread for the shuffle version).

```
Template <valT, 𝕀, fill>
WarpReduce_ShMem( fSum,rSum, ⊕, WS )
if (fill)
  WS[-16] = 𝕀;  // Zero out pads
  WS[0] = rSum; // Set input to scan
else
  rSum = WS[0]; // Get initial scan
end if
// Warp-Scan (Kogge-Stone layout)
WS[0] = rSum = WS[ -1]⊕rSum;
WS[0] = rSum = WS[ -2]⊕rSum;
WS[0] = rSum = WS[ -4]⊕rSum;
WS[0] = rSum = WS[ -8]⊕rSum;
WS[0] = rSum = WS[-16]⊕rSum;
fSum = sm_S3[S3_last]; // Final Sum
end WarpReduce_ShMem
```

```
U32 WarpScan_Regs_AddU32( U32 inVal )
  U32 outVal = inVal;
  asm volatile
  (
  "{\n\t"   // Local Scope (start)
  ".reg .u32 RB;\n\t" // Reach Back
  ".reg .pred P0;\n\t"// Active(T/F)
  "shfl.up.b32 RB|P0, %0,0x1,0x0;\n\t"
  "@P0 add.u32 %0, RB, %0;\n\t"
  "shfl.up.b32 RB|P0, %0,0x2,0x0;\n\t"
  "@P0 add.u32 %0, RB, %0;\n\t"
  "shfl.up.b32 RB|P0, %0,0x4,0x0;\n\t"
  "@P0 add.u32 %0, RB, %0;\n\t"
  "shfl.up.b32 RB|P0, %0,0x8,0x0;\n\t"
  "@P0 add.u32 %0, RB, %0;\n\t"
  "shfl.up.b32 RB|P0,%0,0x10,0x0;\n\t"
  "@P0 add.u32 %0, RB, %0;\n\t"
  "}\n\t"  // Local Scope (stop)
  : "+r"(outVal)  // ASM operands
  );
  return outVal;
end WarpScan_Regs_AddU32
```

Figure 6.12: *`WarpReduce` and `WarpScan` methods*: These methods are implemented using the Kogge-Stone adder, Hillis-Steele variant. The left panel contains code that shares results across threads using shared memory. The right panel contains Kepler-only code that shares results across threads using registers via the .PTX shuffle command. Note: The **volatile** key word used when declaring the warp-scan array pointer into shared memory is very important for correct behavior. Without it, the CUDA compiler will optimize away some of the required instructions.

`WarpScan` and `WarpReduce` are the fastest known GPU methods for scanning and reducing an entire data warp in parallel. Their high performance comes from having low depth (5 stages = $\log_2(32)$), from having few instructions per stage (3 for the shared memory version, 2 for the shuffle version), and by avoiding branching via reach-back padding. Unfortunately, there is a RAW dependency between each pair of instructions in the instruction chain that makes it hard for the SM scheduler to exploit ILP in order to hide stalls. On the other hand, providing multiple parallel thread warps for the SM scheduler to exploit TLP does work to hide stalls.

Total Cycles: Assume n is the run length, S is the amount of time to transfer data between shared memory and registers, and k is the instruction pipeline length. Then, the shared memory version of `WarpScan` takes $[(2S+k)\cdot\log_2 n + 2S]$ total cycles and the shuffle version of `WarpScan` takes $[2k\cdot\log_2 n + k]$ total cycles.

6.6.4 RunUpdate (RU*n*) method

The RunUpdate method (denoted as RU*n*, i.e. RU8 for *n* = 8; this is also known as Fan) updates a short run of *n* elements ($n \in [2..32]$) with a common prefix, which is accumulated into each and every run value. My RunUpdate implementation is fairly straightforward (see Figure 6.13). Since each individual update is independent of all other updates, there are no RAW dependencies between instructions, so this method has strong support for hiding stalls via ILP. This method takes $O(n) = k + n\text{-}1$ total cycles. The ScanUpdate method is not shown (denoted as SU*n*, i.e. SU8 for *n*=8). However, it basically follows a SerialScan immediately with a RunUpdate on the same short run.

```
template< valT, sumT, rL >
RunUpdate( prefix, ⊕, v1, v2, v3, v4, … ) // Add prefix into run
  if (rL >= 1) { v1 = prefix⊕v1; }
  if (rL >= 2) { v2 = prefix⊕v2; }
  if (rL >= 3) { v3 = prefix⊕v3; }
  if (rL >= 4) { v4 = prefix⊕v4; }
  if (rL >= 5) { v5 = prefix⊕v5; }
  if (rL >= 6) { v6 = prefix⊕v6; }
  if (rL >= 7) { v7 = prefix⊕v7; }
  if (rL >= 8) { v8 = prefix⊕v8; }
end RunUpdate
```

Figure 6.13: The RunUpdate method for short runs ($n \leq 8$). Code in light grey gets elided away at compile time.

6.6.5 BlockReduce and BlockScan Methods

I use the RunLoad, RunStore, SerialReduce, SerialScan, WarpScan, and RunUpdate methods as building blocks to implement a 3-level nested reduce or scan over an entire data block. The BlockScan method consists of seven stages (Input, S1-S5, and Output). For BlockScan, the input is one data block of data (*DBS* elements), and the output is one data block of {*inclusive* | *exclusive*} scanned results. The stages are paired as Input and Output, S1 and S5, S2 and S4. The stage S3 is nested within the pair S2 and S4, and stages S2 and S4 are nested within the pair S1 and S5 and the Input and Output stages wrap the rest of the stages S1-S5. The BlockReduce method uses only the first four stages (*Input*, *S1*-*S3*) from BlockScan with the SerialReduce and WarpReduce methods replaced by the corresponding SerialScan and WarpScan methods. For BlockReduce, the input is one data block of data (*DBS* elements), and the output is a single block-sum. For BlockReduce, stage S3 is nested from stage S2 which in turn is nested from stage S1 and the input stage loads the original DBS data elements to be reduced before stage S1 begins its work. To support the sharing of intermediate sums and prefixes across the stages and across the threads, there are also three arrays (S1-S3) kept in shared memory.

***Input*:** For all threads in the thread block, each thread loads a short sequential run of ‹*nWork*› items from memory into registers. The entire thread block (TBS) loads *DBS* data elements from global memory (*DBS* = *nWork*·*TBS*).

***S1* (`SerialScan` #1):** For all threads in the thread block, each thread serially scans, or reduces, its ‹*nWork*› run to a single S2 run-sum. Each thread then stores its S2 run-sum into the S2 array. The entire thread block stores *TBS* S2 run-sums into shared memory (*TBS* = *nWarps*·*WarpSize*).

***S2* (`SerialScan` #2):** For the first thread warp of each thread block (all other warps idle), each thread loads and inclusively scans, or reduces, a short sequential run of ‹*nWarps*› S2 run-sums to a single S3 run-sum. Each thread in the active thread warp stores its S3 run-sum into the S3 array. The active thread warp stores *WarpSize* (32) S3 run-sums.

***S3* (`WarpScan`):** All the threads in the first (active) thread warp then cooperatively warp-scan the S3 run-sums into an inclusive S3 prefix sum. The final block-sum is the last entry of this scanned S3 array.

***S4* (`RunUpdate` #1):** This stage effectively unwinds the nesting of stage S2, each active S2 thread retrieves its missing prefix-sum from stage S3 by reaching back one entry in the scanned S3 array. This exclusive prefix-sum is then accumulated into all the scanned S2 run-sums, and the updated S2 run-sums are stored back into the S2 array as an exclusive run (by reaching back one register entry on output). The active thread warp stores TBS updated S2 run-sums into the S2 array.

***S5* (`RunUpdate` #2):** This stage effectively unwinds the nesting of stage S1, All threads in the thread block load their missing exclusive prefix-sums from the S2 array, as computed in stage S4.

The S2 prefix is first accumulated with the current row-start (missing data row prefix). This common prefix is accumulated into the locally scanned S1 run to achieve globally correct scan results. The final block-sum is then accumulated into the current row-start to prepare for the next data block along the row. The resulting {*inclusive* | *exclusive*} scanned run is then stored back into the S1 array based on the requested type of scan. The entire thread block stores DBS results.

***Output*:** This stage is paired with the *input* stage. For all threads in the thread block, each thread stores its short-scanned run of ‹*nWork*› results from registers back into global memory. The entire thread block stores DBS scanned results into global memory.

***BlockScan* Register Optimization:** For block scans, I preserve the ‹*nWork*› and ‹*nWarps*› runs in registers between the paired stages *S1* and *S5* as well as between the paired stages *S2* and *S4,* respectively. This approach decreases the required I/Os of the nested Scan-then-Fan pattern used in *BlockScan* from four to two and improves performance at the cost of increased register pressure.

My four-stage `BlockReduce` implementation for the *NO* range checking (`BlockReduce_RC_NONE`) case is shown in Figure 6.14. The code for my other three range checked versions of `BlockReduce (*_RC_START, *_RC_STOP, and *_RC_BOTH)` as required by my Row DASk are not shown. The code is quite similar with only minor differences in the `RunLoad` and `RunStore` methods to support range checking.

BlockReduce Method

```
template< valT, sumT, … , nWarps, nWork, 𝕀 >
BlockReduce_RC_NONE( blockSum, inPtr, tid, ⊕
                     S1_store, S1_run, S2_store, S2_run, S3_base )
  // In parallel across all threads in entire thread block
  // INPUT: Load short sequential <nWork> run
     // Convert between warp & sequential view
  RunLoad_RC_NONE<valT, WarpSize, 0, nWork >( inPtr, S1_v1, S1_v2, ... );
  RunStore_RC_NONE<valT,S1_warpPad,0u,nWork>( S1_store, S1_v1, S1_v2, ... );
  RunLoad_RC_NONE<valT, 1u, 0u, nWork>( S1_run, S1_v1, S1_v2, ... );
  // S1: Serial Reduce <nWork> run
  SerialReduce<valT, sumT, nWork>( S1_runSum, ⊕, S1_v1, S1_v2, ... );
  S2_store[0] = S1_runSum;  // Store S1 runSum in S2 array
  if (nWarps >= 2u) { __syncthreads(); } // Barrier
  if (tid <= WarpSize)  // In parallel across all threads in first warp
    // S2: Load & Serial Reduce <nWarps> run of S1 run-sums
    RunLoad_RC_NONE<valT, 1u, 0u, nWork>( S2_run, S2_v1, S2_v2, ... );
    SerialReduce<valT, sumT, nWork>( S2_runSum, ⊕, S2_v1, S2_v2, ... );
    // S3: Load & Warp-Reduce S2 run sums
    volatile valT * S3_warpPtr = (volatile valT *)&(S3_base[S3_first+tid]);
    WarpReduce<valT,sumT,𝕀,WarpSize,true,true>( S3_warpPtr, S2_runSum, ⊕ );
  end if
  if (nWarps >= 2u) { __syncthreads(); } // Barrier
  // In parallel across all threads in entire thread block
  // Grab final block-sum
  blockSum = S3_base[S3_last];
end BlockReduce
```

Figure 6.14: *BlockReduce* code outline. The *BlockReduce* (bottom panel) is done in four stages. **INPUT**) Each thread loads a run of ‹*nWork*› data elements. **S1 –** Each thread *serial reduces* its run. **S2** – Each thread in the first warp loads and *serial reduce* a run of ‹*nWarps*› S1 run-sums. S3 – The first warp cooperatively *Warp-Reduces* the data warp of S2 run-sums down to the final block sum.

My seven-stage `BlockScan` implementation for the *NO* range checking (`BlockReduce_RC_NONE`) case is shown in Figure 6.15. As before, the other 3 range checked versions are not shown. Figure 6.15 consists of two panels. The upper panel shows how to setup various constants, pointer and indices that represent the three shared memory arrays (S1-S3) used by `BlockScan`. The lower panel contains the actual code for the seven-stage `BlockScan` method.

Scan Setup

```
// Constants
TBS = nWarps*WarpSize;              DBS = nWork*TBS;      DWS = nWork*WarpSize;
S1_Stride = S1_PAD+WarpSize;        S1_size = …;
S2_Stride = S2_PAD+WarpSize;        S2_size = …;
HalfPad = WarpSize/2;               S3_First = 1u + HalfPad;
S3_Stride = S3_First + WarpSize;
S3_Last  = S3_Stride-1u;            S3_size = …;
S1_base = 0u;  S2_base = S1_size;  S3_base = (S1_size+S2_size);

// S1-S3 Arrays (in Shared Memory)
__shared__ valT sm_S1[(S1_size+S2_size+S3_size)];

warpRow = threadIdx.y;    // Which Warp in thread block
warpCol = threadIdx.x;    // Which thread in warp
tid     = (warpRow*WarpSize) + warpCol; // Unique Thread ID in thread block

// Zero S1-S3 arrays to identity
SetWarpArrayFixed< … >( sm_S1, 𝕀, warpRow, warpCol );

// Load initial row-start (prefix)
rowStart = rowStarts[blockIdx.y];

// Setup S1-S3 offsets & pointers
IO_off      = (warpRow*DWS) + (S1_base+warpCol);
S1_storeOff = S1_base + RakePow2<WarpSize, 1     , S1_PAD>( IO_off );
S1_runOff   = S1_base + RakePow2<WarpSize, nWork , S1_PAD>( nWork*tid );
S2_storeOff = S2_base + RakePow2<WarpSize, 1     , S2_PAD>( tid );
S2_runOff   = S2_base + RakePow2<WarpSize, nWarps, S2_PAD>( nWarps*tid );
S3_warpOff  = (warpRow*S3_stride)+(S3_base+S3_first+warpCol);

S1_store = &(sm_S1[S1_storeOff]);
S1_run   = &(sm_S1[S1_runOff]);
S2_run   = &(sm_S1[S2_runOff]);
S3_warp  = (volatile valT *)&(sm_S1[S3_warpOff]);

...
...
```

BlockScan Method

```
template< valT, sumOp, BlockSize, WarpSize, WPT, 𝕀 >
BlockScan( out, in, rowStart, tid, ⊕, … ) // … → S1-S3 offsets & pointers
  // In parallel across all threads in entire thread block
  // INPUT: Load run of <nWork> data from global memory
      // Convert from coalesced warp by warp layout to sequential layout
  RunLoad <valT, WarpSize,  nWork>( in, S1_v1, S1_v2, S1_v3, ... );
  RunStore<valT, S1_stride, nWork>( S1_store, S1_v1, S1_v2, S1_v3, ... );
  RunLoad <valT, 1,         nWork>( S1_run, S1_v1, S1_v2, S1_v3, ... );

  // S1: Serial Scan short run of nWork elements
  SerialScan<valT, sumOp, nWork>( ⊕, S1_v1, S1_v2, S1_v3, ... );
  S1_runSum = …;                   // Save S1 run-sum
  sm_S1[S2_storeOff] = S1_runSum; // Store S1 run-sum in S2 array

  if (nWarps >= 2u) { __syncthreads(); } // BARRIER
  if (tid < WarpSize)  // In parallel across all threads in first warp
    // S2: Load & Serial Scan short run of nWarps S1 run-sums
    RunLoad< valT, 1, nWarps >( S2_run, S2_v1, S2_v2, S2_v3, ... );
    SerialScan<valT, sumOp, nWarps>( ⊕, S2_v1, S2_v2, S2_v3, ...);
    S2_runSum = …;                   // Save S2 run-sum

    // S3: Warp Scan (scan S2 run-sums)
    WarpScan<valT,𝕀, true>( S2_runSum, ⊕, S3_warp );
    // S4: Run Update S2 run with S3 prefix
    S3_prefix = S3_warp[-1]; // Grab S2 prefix (Reach Back)
    RunUpdate<valT,runLen>( S3_prefix, ⊕, S2_v1, S2_v2, S2_v3, ... );
       // Store exclusive scanned results in S2 array
    RunStore< valT, 1, nWarps >( S2_run, S3_prefix, S2_v1, S2_v2, … );
  end if
  if (nWarps >= 2u) { __syncthreads(); } // BARRIER
  // In parallel across all threads in entire thread block
  // S5: Run Update S1 run with S3 prefix (& current rowStart)
    // Carry: Also sum final block-sum into current row-start
  blockSum = S3_warp[S3_Last];
  S2_prefix = rowStart⊕sm_S1[S2_storeOff];
  rowStart = rowStart⊕blockSum;
  RunUpdate<valT, sumOp, nWork>( S2_prefix, ⊕, S1_v1, S1_v2, S1_v3, ... );
    // Store {exclusive | inclusive} scanned results in S1 array
  if (bExclusive)
    RunStore<valT, 1, nWork>( S1_run, S2_prefix, S1_v1, S1_v2, ... );
  else
    RunStore<valT, 1, nWork>( S1_run, S1_v1, S1_v2, S1_v3, ... );
  end if
  // OUTPUT: Store scanned run of nWork results
      // Convert from sequential to coalesced view
  RunLoad <valT, S1_stride, nWork>( S1_store, S1_v1, S1_v2, ... );
  RunStore<valT, WarpSize,  nWork>( out, S1_v1, S1_v2, S1_v3, ... );
end BlockScan
```

Figure 6.15: The *BlockScan* (bottom panel) uses a nested *Scan then Fan* pattern in registers to scan an entire data block in 7 stages. The *Top Panel* shows how I setup & pre-compute the S1-S3 view offsets & pointers to support the *BlockScan* method. The *Bottom Panel* contains the pseudo-code for *BlockScan*.

Total Cycles: The resulting total cycle's formula for both *BlockReduce* and *BlockScan* are quite complex. They basically sum up all the cycles used by each helper method across all 4 or 7 stages while including some extra cycles required for additional transition instructions to transfer intermediate results between stages. I summarize the total cycles required for *BlockReduce* and *BlockScan* moving forward as BR_{DBS}‹*nWarps*, *nWork*› and BS_{DBS}‹*nWarps*,*nWork*›, respectively.

BlockReduce and BlockScan Issues: Implementing both methods resulted in the discovery of the following four issues:

1) Each of the S1–S3 arrays require two views into its data (*store* = warp, *run* = sequential).
2) Each stage of the nested method [S1-S3] requires its own shared memory array for communicating results between threads in the thread blocks which may constrain occupancy.
3) Accessing shared memory sequentially may result in *k*-way bank conflicts.

4) Barrier synchronization is required for correct results when sharing data across multiple thread warps.

See Section 6.7 for more details on issues 1-3; issue 4 is discussed in the next section.

***Barrier Synchronization*:** Both `BlockReduce` and `BlockScan` need to communicate and coordinate data across warps within each thread block. For instance, the input to stage S2 is the S1 run-sums from multiple warps. Since the order in which the SM hardware scheduler can call each thread warp is not deterministic, a *barrier* (`syncthreads`; see Section 2.2.3) is needed after stage S1 to force each thread warp to wait for all other thread warps to complete. Similarly, I need another barrier after stage S3 before all warps can safely extract the final block-sum from the last column in the S3 array.

***Constraints on Occupancy*:** Both methods preserve runs in registers across stages and communicate run-sums between warps in shared memory. Because these methods consume a lot of registers and shared memory, both architectures, Fermi and Kepler, quickly reach constraints on occupancy. For better performance, the programmer must take these constraints into account when choosing the initial CTA grid-size launch parameters. If the grid work load does not evenly divide the number of SMs on the current GPU, then left-over SMs must work on their data rows while the rest sit idle, this hurts performance. I discuss this issue more fully in Section 6.7.3.

Benefits: My `BlockReduce` and `BlockScan` methods support sequential access at the second CTA level of threads within a thread block and do this by partitioning a data block into short sequential runs and distributing those runs across all threads. Both methods are implemented using a three level nested pattern in seven (or four) stagesin seven (or four) stages [S1-S3] based on divide and conquer recursion. The `BlockReduce` method reduces the entire data block to a single block-sum. The `BlockScan` method scans the entire data block into a local prefix sum. Both methods supports experiments on finding an optimal combination of TLP and ILP using the *nWarps* and *nWork* template parameters. Both parameters are created as part of initializing and launching the `GPU_Reduce` and `GPU_Scan` kernels and then passed through into the `BlockReduce` and `BlockScan` methods. The *nWork* parameter controls the amount of work per thread in each data block, this shows up in the code as the manual loop unrolling of multiple data elements that the GPU programmer must write to support the desired reduce or scan behavior for the Input (and Output) and S1 (and S5) stages. The *nWarps* parameter controls the amount of thread warps per thread block, this shows up in the code when initializing the S1-S3 storage array pointers for each thread and then the manually loop unrolling for the S2 (and S4) stages. For better performance, both methods need to mitigate issues related to coalescence, bank conflicts, and constraints on occupancy, as is described in Section 6.7.

Limitations: Both my `BlockReduce` and `BlockScan` methods consume many per-thread registers and shared memory resources. Such high consumption can constrain occupancy, limit TLP, and reduce the scheduler's ability to hide latency. Furthermore, both methods use two barriers per data block, which can also slow performance and create ILP scheduling bottlenecks. Both methods are quite complex and specific to reduce and scan operations and thus are not easily reused for other GPU kernels when solving other problems.

6.6.6 GPU *Reduce* and *Scan* Primitives

In this section, I give a high level overview of the implementations for the Reduce and Scan primitives and the kernels that make up each primitive, and then I explain how to use my Row DASk as part of implementing those GPU kernels.

***Reduce*:** My Reduce primitive on the GPU follows the Run-Reduce parallel pattern using two kernels: `GPU_Reduce` and `GPU_SumRows`. My `GPU_Reduce` kernel does most of the work by reducing *n* inputs to *r* row-sums, row by row and block by block, using the `BlockReduce` method on each data block. The kernel then accumulates the resulting block-sums for each data block into a row-sum. The `GPU_SumRows` kernel reduces the *r* row-sums to the final-sum using a single instance of `BlockReduce` on a single thread block. The reduce primitive across both kernels results in $\left\lceil\frac{n}{32}\right\rceil+r+\left\lceil\frac{r}{32}\right\rceil+1 = O(n+r)$ total I/Os. If we assume r is much less than n ($r \ll n$,), then the Reduce primitive requires a little bit more than one global I/O transfer per data warp.

***Scan*:** My Scan primitive on the GPU follows the Reduce-than-Scan parallel pattern using three kernels:`GPU_Reduce`, `GPU_SumsToStarts`, and `GPU_Scan`. The `GPU_Reduce` and `GPU_SumsToStarts` kernels are used together to generate missing row-prefixes for each data row. These row-prefixes, which I call *row-starts*, are then consumed by the `GPU_Scan` kernel to generate globally correct scan results. The `GPU_Reduce` (described in the previous paragraph) does about one-third of the total work involved in running the Scan primitive. The `GPU_SumsToStarts` kernel exclusively scans *r* row-sums into *r* row-starts using a single instance of `BlockScan` on a single thread block. My `GPU_Scan` kernel does about two-thirds of the work by partitioning *n* inputs into *r* data-rows and then scanning each row, data block by data block, using the `BlockScan` method. `BlockScan` generates a block of locally scanned results. The missing row-start (row-prefix) is then summed into each local result, and the globally correct scan results thus obtained are then output. After each data block has been scanned, the resulting block-sum is accumulated into the current row-start to make the row-start correct for the next data block along the row. The Scan primitive across all three kernels results in $\left\lceil\frac{3n}{32}\right\rceil + 2r + \left\lceil\frac{2r}{32}\right\rceil$ $= O(n+r)$ total I/Os. If we assume that *r* is much less than *n* ($r \ll n$), then the Scan primitive requires a little bit more than three global I/Os per data warp on average.

Matching Data Rows Issue: Since the `GPU_Scan` kernel consumes scanned row-starts (as row-prefixes) that were originally generated as row-sums by the `GPU_Reduce` kernel, then both kernels must process the exact same data rows for correct results. To ensure these data rows match up across kernels, I use the same underlying Row DASk for both kernels, and I also use the exact same CTA layout parameters across both kernels.

***Row* DASk:** I use my high-level Row DASk (introduced in Chapter 5) for both the `GPU_Reduce` and `GPU_Scan` kernels as it supports data partitioning into data rows and data blocks and processes blocks sequentially along each row. This DASk also supports warp-alignment and automatic load balancing of data across the grid rows. This DASk also provides setup and support for a range-check pattern [‹FIRST?›, ‹MIDDLE*›, ‹LAST?›], which pushes expensive range checks out of the middle section and into the first and last data blocks. This approach amortizes the range check costs over the large middle section. The initial setup also helps support full coalescence by aligning the input array to a data warp boundary.

There are three main downsides to using my *Row by Row* DASk:

1) There are higher one-time setup costs that need to be amortized across large numbers of data blocks. Thus, this skeleton tends to be slower for small input sizes ($n < 10^6$) than other simpler GPU kernels.
2) The programmer needs to generate four very similar but slightly different versions of the `BlockReduce` and `BlockScan` methods to support the four different range checking interval types. Generating these increases the likelihood of cut and paste style errors.
3) The small grid sizes used ($r < 1{,}000$) tend to make performance sensitive to whether the *workLoad* represented by the grid evenly divides across the SMs on the GPU card.

A high-level outline of my `GPU_Scan` method on the Row DASk is shown in Figure 6.16.

```
template< valT, sumT, 𝕀, bExclusive, logWarpSize
          BlockX, nWarps, nRows, nWork, S1_PAD, S2_PAD >
GPU_Scan( outVals, inVals, inRowStarts, start, stop, startDest, ⊕ )
  // Setup Row by Row Skeleton
  ...  // bid, tid, TBS, DBS, dataOff, startRun, stopRun, rowStride, ...
  LoadBalance<…>( … );
  SetRangePattern<…>( … );  // <FIRST?><MIDDLE*><LAST?> or <BOTH?>

 User Setup (Setup S1-S3 offsets & pointers)
 … // See figure 6.18 (Top Panel)
 …
 // Initialize Row Start prefix
 rowStart = inRowStarts[bid];
  if (rcBoth) // <BOTH?> case => Range check [start, stop]

 BlockScan_RC_BOTH< valT, sumT, DBS, WarpSize, nWarps, nWork, 𝕀 >
   (
     outVals, inVals, rowStart, tid, ⊕, dataOff, start, stop,
     S1_storeOff, S2_storeOff,
     S1_storePtr, S1_runPtr, S2_runPtr, S3_warpPtr
   );

  end if
  if (rcFirst) // <FIRST?> case => Range check [start, …)

 BlockScan_RC_START< valT, sumT, DBS, WarpSize, nWarps, nWork, 𝕀 >
   (
     outVals, inVals, rowStart, tid, ⊕, dataOff, start,
     S1_storeOff, S2_storeOff,
     S1_storePtr, S1_runPtr, S2_runPtr, S3_warpPtr
   );

    dataOff = dataOff + DBS;  // Move to next block along row
  end if
  // Scan all safe data blocks in row (in parallel across all threads)
  while (dataOff < stopRun)  // <MIDDLE*> case => No range checking

 inPtr  = inVals[dataOff+IO_idx];
 outPtr = outVals[dataOff+IO_idx];
 BlockScan_RC_NONE< valT, sumT, DBS, WarpSize, nWarps, nWork >
   (
     outPtr, inPtr, rowStart, tid, ⊕,
     S1_storeOff, S2_storeOff,
     S1_storePtr, S1_runPtr, S2_runPtr, S3_warpPtr
   );
    dataOff = dataOff + DBS;  // Move to next block along row
  end while
  if (rcLast) // <LAST?> case => range check (…, Stop]
 BlockScan_RC_STOP< valT, sumT, DBS, WarpSize, nWarps, nWork, 𝕀 >
   (
     Out, In, rowStart, tid, ⊕, dataOff, stop,
     S1_storeOff, S2_storeOff,
     S1_storePtr, S1_runPtr, S2_runPtr, S3_warpPtr
   );

 end if
end GPU_Scan
```

Figure 6.16: `GPU_Scan` code outline. Changes from the `Copy_Row` DASk shown in purple. There are 4 different user snippets (`BlockScan_*`) but they are all the same except for range checks on loads and stores between global memory and registers.

My `GPU_Reduce` method (not shown) is similar to the `GPU_Scan` code but with four main differences.

1) The `GPU_Reduce` kernel function interface replaces the *outVals* and *inRowStarts* parameters with an *outRowSums* parameter.
2) On Initialization, each thread block sets its row-sum to identity.
3) All calls to `BlockScan` methods are replaced by equivalent calls to `BlockReduce` methods, followed by a simple sum statement to accumulate the block-sum into the current row-sum.
4) On finalization, the first thread in the thread block writes out the final row-sum to the row-sums array.

Template Parameters: The following template parameters (see Table 6.6) are used to support genericity and experimentation:

Parameter	Explanation
valT	This parameter is a place holder for the underlying data type to reduce or scan (U32 for instance).
sumT	This parameter declares the type of the binary operator $\oplus$ used for summation (usually as a C++ functor).
$\mathbb{I}$	This parameter is the identify value for the sum operator and is also used as a default value for out of range loads.
bExclusive	This parameter defines whether an exclusive {*true*} or inclusive {*false*} scan is wanted.
logWarpSize	This parameter is the base 2 logarithm of the fixed number of threads per warp, $5 = \log_2(32)$.
BlockX and *nWarps*	These parameters specify the fixed number threads per thread block as a 2D layout (*TBS*=*BlockX*·*nWarps*) (*TBS*=128). I always set *BlockX* = *WarpSize* (32), so that *nWarps* correctly refers to the number of warps per thread block.
nRows	This parameter is the fixed number of data rows (*r*) from the previous discussion and is also the fixed number of thread blocks in the 1D CTA grid {aka *GridSize.y*}.
nWork	This parameter is the fixed amount of data elements per thread to load and scan (or reduce) in each fixed-size data block.
S1_PAD and *S2_PAD*	These parameters specify whether the "pad and rake" technique (see section 6.7.2) to avoid bank conflicts needs to be done for the stages S1 & S2 respectively. Both pads are set to a value of {0|1}, Zero (0) indicates no padding and raking is needed, One (1) indicates that the appropriate S1 or S2 store & run pointers should be padded and raked to avoid bank conflicts.

Table 6.6: `GPU_Reduce` / `GPU_Scan` Template Parameters

Function Parameters: The following function parameters (see Table 6.7) are used to reduce or scan the input array into an output array of *r* column sums or *n* scanned results respectively:

Parameter	Explanation
outVals	(Scan only) This parameter is the output array that receives *n* scanned results.
outRowSums	(Reduce only)This parameter is the output array that receives *r* row-sums.
inVals	(Scan and Reduce) This parameter is the input array containing the data elements.
inRowStarts	(*Scan* only)This parameter is the input array containing the missing row prefixes for each data row for globally correct scan results.
[*start*, *stop*]	These parameters identify the range of the input array *inVals* to scan or reduce. The input size (*n*) can be computed from the range as *n*=*stop*-*start* +1.
⊕	This parameter is the binary operator (functor) used to sum data.

Table 6.7: `GPU_Reduce` / `GPU_Scan` Function Parameters

CTA Parameters: The following Grid and Block layout parameters are chosen to support TLP:

Parameter	Explanation
Thread Block Size (*TBS*)	This parameter is chosen as a 2D block of threads [128=‹32,4,1›] to take advantage of Thread Level Parallelism (TLP) and support multi-issue on each SM.
Grid Size (GS)	This parameter is chosen as a 1D grid ‹1,224,1› (for a GTX Titan, 224 = 14 SMs per GPU times 16 concurrent blocks per SM to keep each SM busy and to load balance the concurrently running thread blocks evenly across all the SMs on the GPU card.

Table 6.8: `GPU_Reduce` / `GPU_Scan` CTA Parameters

Total Cycles: I can express total cycles in terms of previously defined quantities: *r* data rows (and thread blocks), *c* data blocks per row, *DBS* = *Data Block Size*, *TBS* = *Thread Block Size*. I assume that each `BlockReduce` and `BlockScan` takes BR_{DBS}‹*nWarps*, *nWork*› and BS_{DBS}‹*nWarps*, *nWork*› total cycles respectively.

For the Reduce primitive, the `GPU_Reduce` kernel using *p* SM cores takes $\lceil r/p \rceil$(*c*·BR_{DBS}‹*nWarps*, *nWork*›) total cycles, and the `GPU_SumRows` kernel using only one SM core takes

BR_{DBS}‹*nWarps*, *nWork*› total cycles. For the Scan primitive, the `GPU_Reduce` kernel is as above, the `GPU_SumsToStarts` kernel using only one SM core takes BS_{DBS}‹*nWarps*, *nWork*› total cycles, and the `GPU_Scan` kernel using *p* SM cores takes $\lceil r/p \rceil$($c{\cdot}BS_{DBS}$‹*nWarps*, *nWork*›) total cycles.

Benefits: My full Reduce and Scan implementations inherit all of the following benefits of the Row DASk:

- Achieves I/O throughput similar to the Copy kernel despite the increased complexity of the Reduce and Scan kernels.
- Partitions data into data blocks and then automatically load balances the data blocks across the *r* data rows in the grid with support for range checking.
- Properly range checks all data against the range [*start*, *stop*] while pushing and amortizing range checks into the first and last data blocks.
- Warp aligns starting data offsets [0, 32, 64, …] to support coalescence.

In addition, The Reduce and Scan primitives have the following benefits:

- Supports generic data types and value types
- Supports experiments on TLP and ILP via the *nWarps* and *nWork* parameters;
- Avoids or mitigates bank conflicts using the *S1_PAD* and *S2_PAD* template parameters (see Section 6.7.2)

Limitations: As one would expect, my Reduce and Scan implementations also inherit the limitations of the Row DASk, including the following:

- High setup costs (load balancing, range checking, zeroing arrays) that need to be amortized across lots of data blocks
- Having to write four slightly different versions of code to handle the four different range checks
- Increased register pressure (load balancing, range checking variables).
- Extra branches to handle the four range check cases [‹FIRST?›, ‹MIDDLE*›, ‹LAST?› or ‹BOTH?›]. This is especially true for the while loop for the ‹MIDDLE*› case gets called once per data block. Fortunately, my Row DASk implementation was written to avoid divergent branching.

The 3-level Reduce and Scan kernels, with nested `BlockReduce` and `BlockScan` methods, have additional disadvantages:

- High register use due to multiple work items per thread
- High shared memory use for conversion and stage arrays (S1, S2, S3.)
- Constraints on occupancy caused by register and shared memory limits
- Lots of template parameters, which can confuse the programmer. For example, once the *nWork* and *nWarps* parameters have been chosen, the programmer still needs to correctly pick the

corresponding *S1_PAD* & *S2_PAD* template values if they want to avoid bank conflicts when loading short sequential runs from shared memory.

6.7 Reduce and Scan Implementation Issues

In this section, I show how to mitigate issues of conversion between warp and sequential views, *k*-way bank conflicts, and constraints on occupancy.

6.7.1 Conversion between Warp and Sequential Views

For high throughput to global memory, coalescence must be used, which requires a warp by warp access pattern, and thus a thread wants to access data with a stride equal to one warp (*stride* = 32). However, to support non-commutative summation, consecutive elements cannot be reordered since reordering implies a sequential access pattern, and thus a thread needs to access data with a stride equal to one element (*stride* = 1). To resolve this apparent conflict, I convert back and forth between these two different views of data using a shared memory array. This is a classic space versus. time trade-off.

Table 6.9 shows how to setup offsets and pointers into the shared memory conversion array for each view (warp and sequential).

***Warp* View**
`nWork=4, BankSize=32, Stride=32`
`--------------------------------`
`00000000000000000000000000000000`
`11111111111111111111111111111111`
`22222222222222222222222222222222`
`33333333333333333333333333333333`
	***Sequential* View**
	`nWork=4, BankSize=32, Stride=1`
	`--------------------------------`
	`01230123012301230123012301230123`
	`01230123012301230123012301230123`
	`01230123012301230123012301230123`
	`01230123012301230123012301230123`
	Results in 4-way bank conflicts

`DWS = nWork*WarpSize; // Data Warp Size`
`// Store pointer (Warp view)`
`S1_storeOff = (warpRow*DWS)+warpCol;`
`S1_storePtr = &(sm_S1[S1_storeOff]);`
	`DWS = nWork*WarpSize; // Data Warp Size`
	`// Run pointer (Sequential view)`
	`S1_runOff = (warpRow*DWS)+(nWork*warpCol);`
	`S1_runPtr = &(sm_S1[S1_runoff]);`

`// IN: Convert between Warp & Seq. Views`
`RunLoad<…,Stride=32>(in,inOff,v1,v2, …);`
`RunStore<…,Stride={32|33}>(S1_storePtr, …);`
`RunLoad<…,Stride=1>(S1_runPtr,v1,v2, …);`
	`// OUT: Convert between Seq. & Warp Views`
	`RunStore<…,Stride=1>(S1_runPtr,v1,v2, …);`
	`RunLoad<…,Stride={32|33}>(S1_storePtr,…);`
	`RunStore<…,Stride=32>(out,outOff,v1,v2,…);`

Table 6.9: Convert between *Warp* & *Sequential* views using a shared memory array. Alternating colors indicate alternating threads and their matching sequence of 4 work elements [0123] for each view.

For input, there are three steps required to convert between the warp and sequential views:

1) ‹*nWork*› data warps are loaded from global memory into registers using the warp by warp view (*stride* = 32) of data for high coalesced throughput.
2) ‹*nWork*› data warps are stored from registers into the shared memory array using the *warp by warp* view (*stride* = {32|33}, 33 to avoid bank conflicts).
3) A short sequential run of ‹*nWork*› elements is loaded from shared memory into registers using the sequential view (*stride* = 1).

For output, converting between sequential and warp views is similar, just reverse the input sequence and swap loads with stores.

Converting between the two views supports fully coalesced throughput to global memory as well as sequential access for non-commutative summation. The conversion comes at the performance cost of two extra shared memory accesses (on input and again on output.) This conversion requires a shared memory storage cost equal to the number of elements in the fixed-size data block, *O*(*DBS*), where DBS = *nWork*·*TBS*. Combining this with the memory used by the other stages results in a total shared memory requirement for `BlockReduce` and `BlockScan` of *O*(*DBS*+*TBS*+49). (The magic number "49" represents the size of the S3 array used in stage S3, to support a warp-reduce or warp-scan on a single data warp using the Kogge-Stone adder, Hillis-Steele variant.) This conversion is just for the outermost level of the 3-level nested reduce or scan (AKA the Input and Output stages). Unfortunately, using shared memory to transfer short sequential per-thread runs between shared memory and registers (or vice versa) can result in *k*-way bank conflicts, which I discuss next.

6.7.2 Mitigating Bank Conflicts

Transferring short sequential runs of *n* elements between shared memory and registers can cause *k*-way bank conflicts ($1 \leq k \leq n$). Look at the sequential view panel in Figure 6.22 again. Note that the individual threads start their respective runs at offsets [0, 4, 8, …, 124]. However, since there are only 32 physical memory banks, the starting bank is equal to **mod**(offset,32). Look at the first column with four zeros. The four threads with warp columns equal to [0, 8, 16, 24] respectively all start on the same bank in shared memory [0=(0*8)%32, 0=(8*4)%32, 0=(16*4)%32, 0=(24*4)%32]. This resulting in a four-way bank conflict. This conflict will be serialized (replaying the load or store instruction 3 more times) for each transfer, increasing total cycles and slowing down performance.

Not all sequential runs result in *k*-way bank conflicts. Only those where the run length and the *WarpSize* (32) are not relatively prime to each other do so. As shown in Table 6.10, all odd numbered run lengths will not encounter bank conflicts on transfers.

RL	*k*-way	Pad	RL	*k*-way	Pad	RL	*k*-way	Pad	RL	*k*-way	Pad
1	0	0	**9**	0	0	**17**	0	0	**25**	0	0
2	2	1	**10**	2	**3**	**18**	2	**3**	**26**	2	**1**
3	0	0	**11**	0	0	**19**	0	0	**27**	0	0
4	4	1	**12**	4	**3**	**20**	4	**3**	**28**	4	**1**
5	0	0	**13**	0	0	**21**	0	0	**29**	0	0
6	2	**3**	**14**	2	**3**	**22**	2	**3**	**30**	2	**7**
7	0	0	**15**	0	0	**23**	0	0	**31**	0	0
8	8	1	**16**	16	1	**24**	8	**1**	**32**	32	1

Table 6.10: Padding needed for a given *run length* (RL). The *k-way* columns show the number of bank conflicts that occur without padding. The *Pad* columns show the minimal amount of padding needed to avoid bank conflicts. Pad numbers in bold red have an additional issue with some runs being cut in half by the pad columns.

There are four solutions to mitigate bank conflicts, three that I tried, and one that I discovered that CUDA applied on my behalf – 1) live with the conflicts; 2) use run-lengths that are odd numbers; 3) use the Pad & Rake technique on run lengths that are powers of two; or 4) apply CUDA's aligned vector2 or vector4 optimization on run lengths that are exactly [2, 4, 8] in length.

Live with the bank conflicts: Since potential solutions may cost more operations than the serialized replay instructions caused by the bank conflicts themselves, for some runs it may be faster to just live with the *k*-way bank conflict (especially if *k* is small). This is the solution, I use for runs of length 6. Doing nothing results in one two-way bank conflict per transfer and thus a total of six extra transfer replays per run (where 6 = 6 work items·(2 replays – 1 original transfer)).

Use odd-numbered run-lengths: Since odd numbered run lengths are always relatively prime to the *WarpSize* (32), using odd numbered run lengths will avoid all bank conflicts. This is an easy solution.

Use *Pad & Rake* on run lengths that are a power of two: For run lengths that are a power of two [2, 4, 8, …], I use the *Pad and Rake* technique due to Blelloch (Blelloch, 1995) to avoid any bank conflicts. The pad and rake technique is shown in Figure 6.17 and described in the two paragraphs that follow. This technique will not work with even run lengths that are not powers of two due to a *cut-in-half* issue (where at least one sequential run is cut in half by the extra pad columns), which breaks sequential access.

Pad (with 1 extra column)

```
--------------------------------
*0123012301230123012301230123012
3*012301230123012301230123012301
23*01230123012301230123012301230
123*0123012301230123012301230123
0123............................
```

Shift thread runs over one column

No bank conflicts

Rake

```
// Constants
BankSize = 32; // Num. of Memory Banks
PAD = 1;
...
// Rake run offset
runIdx = nWork*tid;
rRow = runIdx/BankSize;      // idx/32
rCol = runIdx & (BankSize-1); // idx % 32

outIdx = rRow*(PAD+WarpSize))+(PAD+rCol);
```

Figure 6.17: The *Pad and Rake* technique to avoid bank conflicts for a run of length 4. The extra pad column per data warp is shown as a red asterisk. Alternating colors indicate alternating threads within the warp and their matching runs of 4 work elements [0123] each.

Pad: The shared memory arrays are padded with extra unused memory columns. Consider the 32 threads within a thread warp (WarpSize = 32), these extra pad columns shift each thread's access pattern over just enough so that each thread's sequential run starts on a different bank in memory and therefor avoids bank conflicts.

Rake: Raking is the process of re-indexing to skip over the extra pad columns. Conceptually, raking works by switching from the original 1D index without padding to a 2D index (rows, cols) and then back to a 1D index with the extra pad columns accounted for.

All in all, the Pad & Rake technique is tricky to get right and requires more operations, but fortunately the raked offsets can be pre-computed and folded into my view conversion pointer initialization.

Apply CUDA' Aligned Vector2 or Vector Optimization: For run lengths that are exactly 2, 4, or 8, CUDA partially mitigate the bank conflicts using an aligned vector2 or vector4 optimization. For these run lengths, CUDA's low-level SASS compiler will optimize two or four 32-bit load (or store) instructions into a single 64-bit or 128-bit load (or store) instruction. This decreases the number of instructions and bank conflicts by 4× (as instructions that never get executed do not cause bank conflicts). This optimization only gets applied when the shared memory access is sequential, aligned to a 64-bit or 128-bit boundary, and run lengths are exactly 2, 4, or 8. Because the data must be aligned to a proper 64-bit or 128-bit boundary, this *aligned vector4* optimization and the Pad & Rake technique are mutually exclusive.

As we will see in the results section (6.??), This aligned vector4 optimization is faster than Pad & Rake on the GTX Titan (Kepler) but slower on the GTX 580 (Fermi). The results section will also show that living with a high number of non-aligned bank conflicts results in poor performance due to the high number of serialized replays for transfer instructions. For loading sequential runs (length = k) that happen to be powers of two, the number of serialized replays caused by the resulting k-way bank conflicts grows quadratically $O(k) = k^2 = k(k\text{-}1) =\{ 0 = 1\cdot 0, 2 = 2\cdot 1, 12=4\cdot 3, 56 = 8\cdot 7, \ldots \}$.

6.7.3 Constraints on Occupancy

In this section, I want to pick my CTA grid size ($r = nRows$) to achieve good performance. Recall from chapter 5.?? That I define workLoad as *workLoad* = *nSMs·nConBlocks*, where *nSMs* is the actual number of SMs (or SMXs) on a given GPU card and *nConBlocks* is the expected number of concurrent thread blocks that can be concurrently running on each SM at the same time while running a given GPU kernel. I want to pick my CTA grid size to be a multiple of the *workLoad* so that all thread blocks in the grid divide evenly across all the SM's with no left-over thread blocks (rows). A partially full last *workLoad* set means that some SM's continue to work while the rest SM's idle. Given a very large gridsize, the poor performance of the last partially full workLoad can be amortized across all the full workLoads. This means that the Block DASk can effectively ignore this issue for large grids. However, Since I am deliberately keeping the number of rows small ($r \leq 1000$) so that `GPU_SumRows` or `GPU_SumsToStarts` kernel can complete using a single thread block (GS = 1), I need to compute the correct workLoad and make my grid size a multiple of this work load for best performance.

Unfortunately, computing the number of concurrent blocks per SM is complicated by various constraints on occupancy (see Section 3.3). Recall that the number of concurrent thread blocks and active thread warps on a GPU are limited not only by direct hardware constraints (‹8, 48› on Fermi, and ‹16, 64› on Kepler) but also by the register pool size (32K on Fermi and 64K on Kepler) and shared memory pool size (48 KB on both Fermi and Kepler) on each SM. With some effort, shared memory use can be predicted ahead of time. Register use by the CUDA compiler is not easily predicted, however. It requires experiments with the "*–verbose*" output compiler flag turned on.

To maximize performance, I iterate on a six-step process in order to choose an initial CTA grid size that respects constraints on occupancy and results in a *workLoad* that evenly divides thread blocks across the number of SMs on each GPU:

1) I compile my kernels with the CUDA "*–verbose*" output flag run on, which reports the expected registers per thread (count) and shared memory (in bytes) used per block. Note: these results are estimates; the runtime hardware may actually use more or fewer registers.
2) From these compiler outputs, I compute the excepted number of concurrent thread blocks that can run at the same time *nExpectBlocks* = **min**(*regBlocks*, *shareBlocks*, {8|16}). {8|16} comes from the max concurrent blocks per SM. The register pool size limits blocks as *regBlocks* = floor({32K|64K}/(*TBS***RPT*)), where{32K|64K} is the register pool size (Fermi or Kepler), *TBS* = threads per block, and *RPT* = registers per thread (from verbose output). The shared memory pool limits blocks as *shareBlocks* = **floor**({16 | 32 | 48KB }/*SBB*), where {16 | 32 | 48 KB } is the size of the shared memory pool and *SBB* = shared bytes per block (from verbose output).
3) From the expected concurrent blocks, I compute my initial work load as *workLoad* = *nSMs·nExpectBlocks*, where the number of SM or SMX cores is GPU card specific. The work load is the minimum number of thread blocks needed to evenly divide the grid of thread blocks across all SM cores on the current GPU device.
4) I choose my actual grid size as a multiple of the *workLoad* to load balance work evenly across the SMs on the current GPU card.
5) I run the code using the chosen CTA (grid size, thread block size) and track performance. If I get slow performance, then either I may have miscalculated or the true register use is higher than the CUDA compiler originally suggested. I try reducing the number of concurrent blocks by {1|2} concurrent blocks and re-measure.
6) I repeat steps 1-5 as needed after I rewrite my code, change my data design layout, or modify my CTA parameters until I am satisfied with the throughput results.

6.8 Results

I focus in this section on experiments involving ILP and TLP to find the best performance results. All tests were performed on a GTX 580 (Fermi) and a GTX Titan (Kepler) using the same host environment (as shown in Figure 6.25).

CPU Hardware: CPU = i7-4770K@3.50 GHz, RAM=12 GB
GPU Hardware: *GTX 580* (16 SMs, 512 SPs, 1.5 GB RAM, 192.4 GB/s peak throughput) *GTX Titan* (14 SMXs, 2,688 SPs, 6.0 GB RAM, 288.4 GB/s peak throughput)
Software: GPU API = CUDA 5.5, C++, IDE = VS 2010, OS = Windows 7, SP1, Pointers = 64-bit
Data: Input size, $n = [2^{10} – 2^{28}]$, in increasing powers of two
Table 6.11: *Reduce/Scan Experiment Environment*

All timings are derived from 101 runs of each test by dropping the first run and averaging the timings for the last 100 runs. *I/O throughput* is measured in gigabytes per second (GB/s) by counting the number of giga-bytes processed and dividing by the average run time in seconds. *Total cycles* (*TC*) are computed as *TC*=*II*/*IPC* by grabbing raw counts (*I*nstructions *I*ssued [*II*], and average *I*nstructions retired *P*er *C*ycle [*IPC*]) from NVidia's NSight Profiler.

In this results section, I focus on two sets of experiments: First experiments on finding the best throughput by varying TLP and ILP parameters, see Section 6.8.1; Second by running experiments to measure the total cycles for some of the more interesting TLP and ILP parameters pairs, see Section 6.8.2.

6.8.1 Throughput

At the end of the chapter introduction, Figure 6.4 shows four plots of throughput results by input size for Reduce (2 kernels) and Scan (3 kernels) on both the GTX 580 and GTX Titan. Four main factors impact performance: TLP, ILP, efficient memory access patterns, and mitigating issues, such as converting views, k-way bank conflicts, and constraints on occupancy

I vary both TLP and ILP using template parameters, ‹*nWarps*, *nWork*›. For example, my *baseline* pair ‹*nWarps*=1, *nWork*=1› uses a single thread warp per thread block and loads and reduces/scans a single work item per thread. I measure throughput for increasing *n* (as powers of two) for both *nWarps* and *nWork* in the range {1..8} resulting in 64 possibilities. I summarize some of the relevant results below.

Baseline ‹1,1›: My baseline case experiments shows the performance achieved when no extra TLP or ILP is applied via my template parameters. The best throughputs are taken from each throughput curve (typically when the input size (n) = 2^{28}). The throughputs for all four baseline cases (Reduce vs Scan on both a GTX 580 and GTX Titan) are summarized in Table 6.12.

***Reduce* (GB/s)**		***Scan* (GB/s)**	
580	**Titan**	**580**	**Titan**
24.24	40.04	33.57	53.71

Table 6.12: *Best Case Summary*

TLP-Focused ‹1-8, 1›: In these experiments (see Table 6.13), I test the impact of TLP by varying the template parameter *nWarps* in the range {1-8} and leaving *nWork* fixed at {1, no extra ILP} for $n = 2^{28}$. On the GTX 580, the best throughput for Reduce and Scan occurs with pair's ‹6, 1› and ‹5, 1› respectively. On the GTX Titan, the best throughput for both Reduce and Scan occurs with pair ‹4, 1›.

	***Reduce* (GB/s)**		***Scan* (GB/s)**	
nWarps	**580**	**Titan**	**580**	**Titan**
1	24.24	40.04	33.57	53.71
2	40.93	59.29	55.60	77.44
3	58.49	90.32	76.20	114.06
4	71.34	**107.85**	90.47	**136.70**
5	84.01	103.71	**104.28**	130.15
6	**91.51**	93.64	94.84	115.39
7	82.34	102.94	89.26	125.67
8	77.05	104.68	81.35	128.57

Table 6.13: *TLP Reduce / Scan Results*

ILP-Focused ‹1, 1-8›: In these experiments (see Table 6.14), I test the impact of ILP by leaving *nWarps* fixed at {1, no extra TLP} and varying *nWork* in the range {1-8} for $n = 2^{28}$. On the GTX 580 the best throughput occurs with the pair ‹1, 6› for both Reduce and Scan. On the GTX Titan the best throughput occurs with the pair ‹1, 8› for both Reduce and Scan.

	Reduce (GB/s)		*Scan* (GB/s)	
nWork	**580**	**Titan**	**580**	**Titan**
1	24.24	40.04	33.57	53.71
2	44.60	72.36	59.84	93.23
3	63.36	95.83	80.60	123.15
4	81.25	124.78	97.54	152.11
5	94.55	122.72	112.45	147.57
6	**108.80**	133.27	**128.37**	151.89
7	98.67	152.28	118.43	168.00
8	100.34	**164.36**	120.08	**175.95**

Table 6.14: *ILP Reduce / Scan Results*

Comparing the results from the TLP and ILP experiments, we see that ILP has a bigger impact and TLP has a lesser impact. Even though, with our copy case studies in Chapter 4, the reverse was the case. These results make sense to me. Recall that the way I wrote my 3-level nested *BlockReduce* and *BlockScan* methods. The outer most stage (*S1*) is driven directly by the *nWork* parameter, with all threads in the thread block reducing (or scanning) a short run of *nWork* elements. While the two-inner most stages (*S2* and *S3*} are driven by the *nWarps* parameter, reducing (or scanning) a short run of *nWarps* elements and then reducing (or scanning) the final data warp. Recall also that in stages *S2* and *S3* and *S4* for Scan) that only a single warp is active, while the rest of the warps in the thread block sit idle. So, the impact of TLP is in effect muted as only one warp out of *nWarps* is doing useful work in these inner stages. Naturally, the programmer would consider reversing the TLP and ILP (inner and outer stages) with the hope of better throughput. However, I have tried this with disappointing results.

Best Throughput: I ran experiments for all 64 combinations of ‹*nWarps*, *nWork*›, each in the range {1–8}, to find the best overall throughputs for Reduce and Scan on the GTX 580 and GTX Titan GPUs. Based on my ILP and TLP test results, I would have predicted the pair's ‹6, 6› and ‹5, 6› would produce the best throughput for Reduce and Scan on the GTX 580. I would have similarly predicted the pair ‹5, 6› to have the best throughput for both primitives on the GX Titan. However, after trying all 64 combinations, I discovered that this was not the case. Other pairs were faster. After thinking about this, I realized that as I varied the ‹*nWarps*, *nWork*› parameters, I also increased register and shared memory usage, which puts constraints on occupancy, making it hard to predict winners. I had to brute force test all combinations to find the best pairs.

The experimental pairs resulting in the best throughput are summarized in Table 6.15 (with best throughputs typically for $n = 2^{28}$.)

	Reduce **(GB/s)**		***Scan*** **(GB/s)**	
	580	**Titan**	**580**	**Titan**
Best Pair	‹3, 6›	‹3, 8›	‹3, 6›	‹5, 4›
Throughput	**172.34**	**220.30**	**163.75**	**220.06**
% of Baseline	711%	550%	488%	410%
% of Copy	98.5%	93.2%	93.6%	93.1%
% of Peak	89.6%	76.4%	85.1%	76.3%

Table 6.15: Best ‹TLP,ILP› Summary.

For run lengths that are powers of two {4,8}, I use the Pad and Rake technique to avoid bank conflicts. For run lengths equal to six, I live with the serialized replays caused by the resulting two-way bank conflicts. In the next section, I describe the results from trying various experiments on mitigating bank conflicts.

Bank Conflicts: In this set of experiments, I explore the impact of bank conflicts (and the resulting serialized replay instructions) on throughput. Figure 6.30 shows GTX Titan Scan throughput results for five different ways to handle bank conflicts:

- Using the Pad and Rake technique, which avoids all bank conflicts
- Letting CUDA apply its *aligned Vector4* optimization on shared memory transfers, which decreases the number of transfer instructions and bank conflicts by 4×
- Using odd-numbered run lengths, which avoids all bank conflicts ‹3, 5›
- Living with a low number of two-way bank conflicts ‹6, 6›
- Living with a high number of non-aligned bank conflicts ‹4, 8› {4-way & 8-way bank conflicts}.

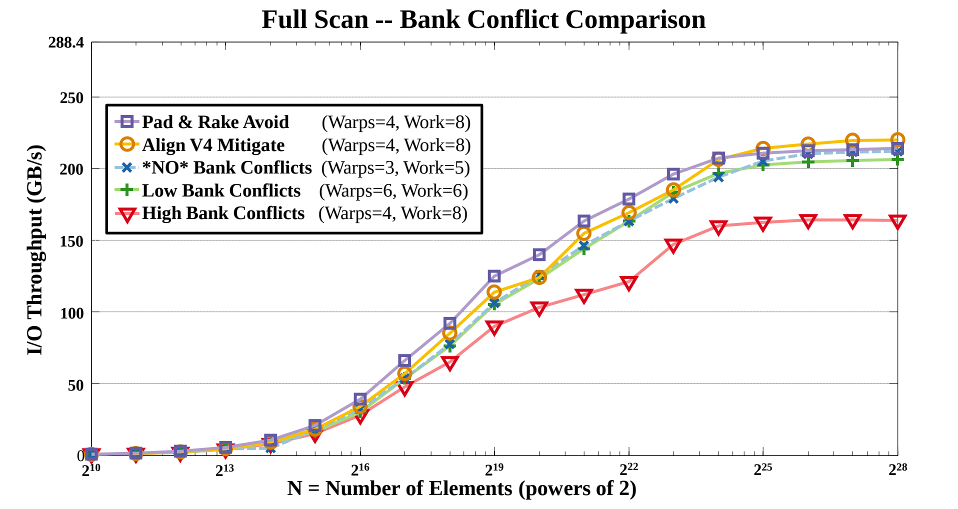


Figure 6.18: Throughput results for five different ways to handle bank conflicts for a full scan on the GTX Titan (Kepler).

The throughput curves (Figure 6.18 above) and in the summary table (Table 6.16 below) of best results capture the throughput results. As can be seen, the first four methods for dealing with bank conflicts are all reasonably good. However, living with a large number of bank conflicts negatively impacts throughput due to the high number of instruction replays caused by serialization. Living with a high number of bank conflicts for the pair ‹4, 8› runs at only about 75% of the throughput of completely avoiding them using the Pad and Rake technique.

BC Results	**Pad & Rake**	**Align V4**	**Odd Runs**	**Low BC**	**High BC**
Pair‹*nWarps*,*nWork*›	‹4, 8›	‹4, 8›	‹3, 5›	‹6, 6›	‹4, 8›
Throughput (GB/s)	214.02	**220.05**	211.90	206.40	**163.50**

Table 6.16: *Mitigating Bank Conflict Results*

Recall that the serialized instruction replays caused by bank conflicts for power of two run lengths grow at a quadratic rate with the run length (0, 2, 12, 56, …). Interestingly, at least on the GTX Titan (Kepler), CUDA's *Aligned Vector4* optimization is faster than the Pad and Rake technique. The opposite is true on the GTX 580 (Fermi). As can be seen from the throughput curves and results, mitigating bank conflicts is well worth the extra effort for my Reduce and Scan primitives.

6.8.2 Total Cycles

In this section, I gather TC results from the NVidia Compute Visual Profiler (on CUDA 5.5) on the GTX Titan for $n = 2^{28}$. The results are summarized in Table 6.17 below. The *instructions issued* (*II*) and *TC* columns are measured as millions of instructions or cycles, respectively. The *instructions* retired *per cycle* (*IPC*) column has a maximum value of 5.0 instructions per cycle on the GTX Titan. The Throughput column (for comparison) is measured in Giga-Bytes per second (GB/s). The type pairs with dark red font have bank conflicts (*V4* = 4.2M, *Low* =10.5M, and *High* = 54.0M), the rest of the rows have no bank conflicts.

Type Pair	*II*	*IPC*	*TC* (*II*/*IPC*)	Throughput (GB/s)
Base‹1 1›	364.9M	0.78	*467.9M*	*53.7*
TLP‹4,1›	304.0M	**1.59**	191.2M	136.7
ILP‹1,8›	**110.1M**	**0.66**	166.9M	176.0
P&R‹5,4›	133.8M	0.98	**136.5M**	**220.0**
V4‹5,4›	127.7M	0.95	**134.4M**	**224.0**
Odd‹3,5›	126.1M	0.89	**141.6M**	**211.9**
Low‹6,6›	128.4M	0.86	**149.3M**	**206.4**
High‹4,8›	172.2M	1.06	162.5M	163.5

Table 6.17: *Total Cycle Results*

There are four significant insights that I see in the total cycles data. First, there appears to be a strong inverse correlation between total cycles and throughput. When total cycles is low, throughput is high (and vice versa). Second, a high number of instruction replays caused by bank conflicts significantly increases total cycles and thus decreases throughput performance. For instance, the **High‹4, 8›** row issues a lot more instructions (including replays) than the **P&R**, **V4**, and **Odd** rows that mitigate or avoid bank conflicts. Third, the combination of both low instructions issued (II) and high instructions retired per cycle (IPC) results in the overall throughput winners. For instance, the **TLP‹4, 1›** row has great instructions retired per cycle (IPC = 1.59) but still performs poorly because it issues so many instructions (II = 304 million). On the other hand, the **ILP‹1, 8›** row has the smallest number of instructions issued (II = 110 million), but still performs poorly because it has the worst instructions retired per cycle (IPC =

0.66). Fourth and finally, the two highest performing rows **P&R‹5, 4›** (Avoid Bank Conflicts via Pad & Rake) and **V4‹5, 4›** (Mitigate Bank Conflicts via the aligned Vector4 optimization) each have the second best instructions issued (II) and the third best instructions retired (IPC) but they result in the lowest total cycles and thus the best throughput.

The formula for total cycles ($TC = II/IPC$) tells me that there are two ways to attempt to improve performance by decreasing TC and thus improving throughput:

1) Decrease instructions issued (II) by trying different algorithms or simplifying code.
2) Increase instructions retired, by increasing ILP via software pipelining and increasing TLP via higher occupancy, to consume as many instructions per cycle (IPC) as possible.

To me, the main lesson is to keep our parallel processing cores (SMs and SPs) and memory controllers as busy as possible. TLP supports such an effort by providing lots of active thread warps for the SM on-core scheduler to switch to when the currently executing warp stalls. ILP supports it by providing lots of independent instructions for the scheduler to harvest to keep the instruction pipeline as full as possible. Fortunately, both approaches are orthogonal. Therefore, both techniques (TLP and ILP) can be used to hide pipeline and I/O stalls and keep the processing cores as busy as possible.

6.9 Conclusion

In this chapter, I introduced Reduce and Scan primitives that require only the associative property (re-grouping) and forbid the commutative property (re-ordering). Non-commutativity implies that short data runs must be summed or scanned in consecutive order. To do this in the 2-level CTA hierarchy of GPU's, I used a 2-level access pattern. At the CTA first level (blocks), I partitioned data using my Row DASk that sequentially marches along each data row, block by block. At the CTA second level (threads), for each data block I transferred short runs of consecutive data between global memory and registers using a conversion array in shared memory.

To make this all work, I had to overcome three main issues that hindered performance:

1) To respect coalescence for better global memory throughput, I actually transferred data between global memory and shared memory using the warp by warp access pattern. I then converted data to a sequential access pattern using shared memory.
2) To avoid bank conflicts when accessing shared memory which can cause serialized replays, I discussed and tested several ways of mitigating bank conflicts: using odd length runs, using the *Pad & Rake* technique on power of two length runs, using *Aligned Vector4's* so that CUDA can decrease bank conflicts by a factor of 2× or 4×, and, finally, just living with a low number of serialized replays caused by bank conflicts.
3) Using the Row DASk results in small grid sizes, which makes my solution sensitive to picking a good work load in six steps that evenly divides the thread blocks in the grid across the SMs. Picking a good work load requires that the programmer understands constraints on occupancy and picks the initial grid size accordingly.

By increasing TLP, increasing ILP, using efficient I/O access patterns, and mitigating bank conflicts, I improved Reduce and Scan performance over my baseline performance. The best performing throughput for the Reduce primitive was up to 7.1× and 5.8× faster than the baselines on the GTX 580 and GTX Titan respectively. The best performing throughput for the Scan primitive was up to 4.9× and 4.2× faster than the baselines.

6.9.1 Limitations

My GPU Reduce and Scan implementations have three main limitations.

1) **Sensitivity to Work Loads:** My method uses small grid sizes ($r < 1000$) which means that if the grid work load is not setup to evenly divide rows across SMs based on the correct number of concurrent thread blocks per SM than performance will suffer.
2) **Potential Copy and Paste Errors:** The amortized range checking pattern used by the Row DASk means that I had to write the same `BlockReduce` & `BlockScan` methods four times with minor differences to support different types of range checking NONE, [*start*, …), (…, *stop*] and [*start*, *stop*]. This means if I find and fix bugs in one of my `BlockReduce` or `BlockScan` methods, I have to remember to make corresponding changes in the other three versions.

3) **Brute-Force Search of Parameter Space:** Due to varying constraints on occupancy as register and shared memory usage increase, searching the parameter space ‹*nWarps*, *nWork*› for the best throughput is currently brute-force.

6.9.2 Future Directions

My implementations achieved a solid percentage of peak throughput on both the GTX 580 (*Reduce* = 89.8% and *Scan* = 85.1%) and GTX Titan (*Reduce* = 79.7% and *Scan* = 77.9%), but perhaps other clever algorithmic improvements, or a ‹*nWarps, nWork*› pair outside the range I explored, code simplification, or leveraging of other GPU features may lead to further performance gains.

Overflow Support: The Reduce and Scan primitives should be rewritten to upscale input data types (32-bit) to larger output data types (64-bit) to handle summations on millions or billions of elements.

Commutative Reduce: For summations, if the commutative as well as associative property is assumed, then the Reduce primitive can be completely rewritten to sum data blocks into per-thread sums and then after all the data blocks have been exhausted, block reduce the per-thread sums to a single row-sum. Reordering data enables direct use of the warp by warp access pattern, thus eliminating the overhead of my current conversion code (between warp and sequential views). Rewriting also eliminates the need for a 3-level nested reduce and barriers. The resulting simplified commutative Reduce kernels should take far fewer instructions, fewer total cycles and thus have higher throughput. Unfortunately, the GPU Scan kernels still requires consecutive access for correct prefix-sum results. Nevertheless, since the `GPU_Reduce` kernel accounts for about one-third of the total work in the full scan primitive, there will be a modest increase in throughput for the Scan primitive when used with a faster commutative `GPU_Reduce` kernel.

6.10 Lessons Learned from Reduce/Scan

In this section, I summarize the lessons learned from this case study. Here is the notation that I typically use. The symbol *n* is used to represent the run length, the warp size, or the input size depending on context. The symbol *k* is used to represent two unrelated concepts 1) the pipelined instruction length

(18-22 cycles on Fermi, 9-11 cycles on Kepler) or 2) alternately the number of conflicting threads in a *k*-way bank conflict. The symbol *S* is used to represent the number of cycles to transfer data between shared memory and registers (40-80 cycles on Fermi, 20-40 cycles on Kepler). The symbol *G* is used to represent the number of cycles to transfer data between global memory and registers (400-800 cycles).

`SerialReduce` and `SerialScan` Lessons:

- For both methods, it is faster to load a short run (length *n*) into registers and then Reduce or Scan on the run stored in registers instead of Reduce or Scan on the run stored in shared or global memory. The in-register based methods take $O(n)$ registers, while the in-memory based methods take $O(1)$ registers.
- Sequential serial reduce or serial scan methods in registers, shared memory, and global memory take $O(k \cdot n)$, $O(S \cdot n)$ and $O(G \cdot n)$ cycles respectively and $O(1)$ registers.
- *Serial Reduce*: For better performance on short runs ($n \leq 8$), use Sklansky's adder layout as the basis for the `SerialReduce` method (see Figure 6.12). This takes $O(n + k \cdot \log_2 n)$ cycles and $O(n)$ registers.
- *Serial Scan*: For better performance on short runs ($n \leq 8$), use Sklansky's adder layout as the basis for the `SerialScan` method (see Figure 6.12). This takes $O([(n/2)+k] \cdot \log_2 n)$ cycles and $O(n)$ registers.
- For longer runs ($8 \leq n \leq 64$), consider using a nested serial reduce or serial scan in batches of 4 or 8 work-items to mitigate register pressure.
- Overloading the summation operator as shown in figure 6.14 prevents the CUDA compiler from de-optimizing away (via unwanted stalls) some of the expected performance.

`WarpScan` and `WarpReduce` Lessons:

- For best parallel performance, Use the `WarpReduce` or `WarpScan` methods to reduce or scan a single data warp using a single thread warp ($n = 32$).
    - On Fermi, use `WarpReduce` or `WarpScan` in shared memory. This takes $O(1) = 3$ registers per thread and takes $O([2S+k] \cdot \log n + 2S)$ total cycles and also requires $O(WarpSize) = 49$ (1+16+32) storage space per active warp (49 data elements per active thread warp for the *warp* array kept in shared memory). For both shared memory methods, the last column of the warp array contains the final warp-sum.
    - On Kepler, use `WarpReduce` or `WarpScan` in registers with the .PTX shuffle command. This takes $O(1)=3$ registers per thread and takes $O([2k \cdot \log n]+k)$ total cycles and requires no shared memory. For both shuffle methods in registers, only the last thread in the warp ends up with the final warp-sum.
- Both methods are expensive due to RAW dependencies between each pair of instructions and the higher cost of shared memory accesses (on Fermi). So only use to reduce or scan the last data warp (32) inside a nested reduce or scan.

`BlockReduce` and `BlockScan` Lessons:

- To support coalescence and high-global memory throughput on input and output, use a warp-by-warp BASk. (see Chapter 5.1.1. for details.)
- To support non-commutative summation, convert the warp-view into a sequential-view to load short sequential runs of *nWork* data elements (see Section 6.7.1. for details on this conversion)
- Mitigate bank conflicts (see Section 6.7.2. for more details)
- To coordinate data communication across thread warps, use barriers to synchronize behavior.
- For increased TLP performance via increased occupancy, write code or design data layouts that minimize register and shared memory usage.
- For increased ILP via reduced total cycles, write code that minimize the total number of instructions, and manually unrolls (software pipelines) multiple work items to avoid hardware pipeline stalls.
- For `BlockScan`: For missing prefixes, reach back one column in shared memory (or in registers) to grab inclusive results from an inclusive prefix sum.
- For `BlockScan`: use the Scan-then-Fan pattern, and preserve runs in registers between paired stages (S1 & S5, S2 & S4) to decrease the total instructions (summations and shared memory accesses).
- Setup two views (pointers) into each of the S1-S3 data arrays. A store view where a single thread stores a single value or run-sum into the array. And a run view where a single thread loads a short sequential run of values from a starting offset or pointer.

Reduce and Scan Kernel Lessons:

- For best throughput, support coalescence.
- For better performance, as part of kernel setup, pre-compute the views (warp-view and sequential-view as pointers or indices) used in the `BlockReduce` and BlockScan methods. Note: storing these view pointers or indices increases register pressure
- To reduce code overhead, use the Row DASk as a starting point.
- Under the assumption of non-commutativity, both Reduce and Scan require consecutive access within runs for correct results. So use the Row DASk to implement the `GPU_Reduce` and `GPU_Scan` kernels. As the Row DASk supports sequential access along each row for the first level of the CTA (thread blocks within a grid).
- Since, the Row DASk requires four bodies of very similar code to properly support amortized range checking, watch out for copy and paste errors in the `BlockReduce` and `BlockScan` methods. If you fix a bug in one version, remember to check and fix the other versions as well.
- To keep the SMs busy and to load balance thread blocks evenly across SMs, pick the number of rows (*r*) to be a multiple of the *workLoad* = *nSMs·nConBlocks* (see section 6.7.3 for details on how to compute the work load).
- For Scan: Make sure the `GPU_Reduce` and `GPU_Scan` kernels both use the exact same data rows for correct scan results. My solution is to have both kernels use the same Row DASk, the same CTA layout, and the same DBS.
- For Scan: For fewer global I/Os (3× instead of 4×), use the Reduce-then-Scan pattern, and avoid the Scan-then-Fan pattern. The Row DASk on both the `GPU_Reduce` and GPU_Scan kernels naturally supports this pattern. From hard personal experience, I advise programmers to avoid using the Block DASk as the base framework for implementing Scan. Using the Block DASk

leads to a recursive Scan-then-Fan pattern, which is not only tricky to get working correctly but, in my experience, it is also quite slow.

General Advice:

- To find the best performance, use both ILP and TLP.
  - For increased TLP, each thread block should contain multiple warps (*nWarps*).
  - For increased ILP, each thread should load and process multiple work items (*nWork*).
- Mitigate bank conflicts in shared memory to avoid costly replays (see section 6.7.2)
  - Avoid bank conflicts using run lengths that are odd numbers.
  - Avoid bank conflicts using the *Pad & Rake* technique for run lengths that are powers of two.
  - Reduce bank conflicts by aligning runs to 64-bit or 128-bit boundaries to let CUDA optimize shared memory access using 64-bit or 128-bit load/store commands (aligned Vector 2 and Vector 4 optimization.
  - Live with low number of bank conflicts such as 2-way bank conflicts. Note: Trying to live with high number of bank conflicts (4-way or higher) may negatively impact performance due to costly replays.
- Decrease Total Cycles to increase throughput (*TC* = *II*/*IPC*)
  - Decrease *instructions issued* (II) by simplifying code, avoiding dependencies between instruction, and software pipelining.
  - Increase *instructions* retired *per cycle* (IPC) by increasing occupancy, minimizing register & shared memory usage.

7.0 Case Study: *k*d-Tree

The *k*d-tree is a spatial partitioning data structure that supports efficient nearest neighbor (NN) searches on a CPU. Although a depth-first search (DFS) *k*d-tree is not a natural fit for GPU implementation, it can still be effective with the right engineering decisions. In my implementation, by bounding the maximum height of the DFS *k*d-tree, minimizing the memory footprint of data structures, and optimizing the GPU kernel code, multi-core GPU NN searches with tens of thousands to tens of millions of points run 20-40 times faster than the equivalent single-core CPU NN searches even after I rewrote the CPU code with the knowledge gained from optimizing the GPU code.

Be aware that this case study is the first algorithm (DFS kd-tree NN searches) I ever adapted to work on a GPU. As a result, I had not yet invented data access skeletons (DASk), so I do not use them in this case study. Also, I was not as familiar with concepts of coalescence, branch divergence, occupancy, and constraints on occupancy and therefore I did not really know how these concepts can impact performance. As a result, most of the parallel performance I do achieve in this case study comes from the first level of the CTA (thread blocks within a grid mapped onto SMs) and engineering decisions to streamline the *k*d-tree search algorithm on both the CPU and GPU. For the second level of the CTA (threads within a thread block mapped onto SPs), threads within each thread warp diverge down different search paths through the *k*d-tree. Since all threads in each warp move in lock-step through the instruction stream, the slowest thread in each warp gates performance for the entire warp.

Unfortunately, this thread divergence eliminates most of the performance advantage of using multiple threads per thread block. As will be seen later in this chapter, experiments show that the best performance occurs for small thread blocks sizes (4-16 threads per thread block).

All my experiments were done on an older GTX 285 GPU card (30 SMs with 8 SPs per SM, 159 GB/s peak throughput for global memory). Since more modern GPUS (GTX 580 and GTX Titan vs. GTX 285) have even fewer SMs (16 and 14, respectively, vs. 30) at the first level of the GPU core hierarchy, I predict that my NN *k*d-tree GPU kernels will perform worse if run on these more modern GPUs.

7.1 Nearest Neighbor (NN) Problem Definitions

The NN problem, which finds the closest point in a point cloud to a specified query point, is important in many areas of computer science, including computer graphics, machine learning, pattern recognition, statistics, and data mining (Shakhnarovich and Indyk, 2005).

Despite its importance and the frequency of its use, there are several NN search problems. In each NN search, input consists of sets of *n* searched points, *S*, and *m* query points, *Q*, and a distance metric in *d* dimensions (such as Euclidean, Manhattan, Chebyshev, or Mahlanobis distance). The Euclidean distance between search point $p \in S$ and query point $q \in Q$ is $dist(p,q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_d - p_d)^2}$. Output consists of the *km* nearest points, where *k* is the number of requested nearest points in the search set *S* for each query point in *Q*. I define six common NN searches (see Table 7.1), the first four of which (in black type) I implement and test. The last two (in gray), I do not.

NN Type	Abbr.	Input	Output
Query Nearest Neighbor	QNN	*S, Q, dist*	*R* = *m* closest points
k Nearest Neighbor	*k*NN	*S, Q, k, dist*	*R* = *km* closest points
All Nearest Neighbor	All-NN	S (== *Q*), *dist*	*R* = *n* closest points
All k-Nearest Neighbor	All-*k*NN	S (== *Q*), *k, dist*	*R* = *kn* closest points
Range Query	RNN	*S, QR, dist*	*R* = varying list
Approximate Nearest Neighbor	ANN	*S, Q, dist*	*R* = *m* closest points

Table 7.1: A Table of six common *nearest neighbor* (NN) searches with typical abbreviations, inputs, and outputs.

For the query nearest neighbor (QNN) search, I find the closest point (k=1) in the search set S for each query point in Q under the distance metric *dist*, this produces the output set R containing m result points. For the 'k' *nearest neighbor* (kNN) search, I generalize QNN to find the k closest points in the search set S for each query point in Q (producing R containing km points). For the all nearest neighbor searches (All NN and All kNN), I assume that the query set Q and search set S are one and the same ($Q = S$). Assuming that $Q = S$ results in a new issue where I need to carefully exclude zero distance results; otherwise, each query point would return itself as part of the search results.

There are a couple of other NN searches that kd-trees can support. However, although I define them, I will not discuss them further. In the range query nearest neighbor (RNN) search, I find all points from the search set S contained in each individual query region belonging to the query set *QR*(*egion*). This approach results in a varying number of result points for each region. Consequently, the final output size is indeterminate. Each individual query region QR_i is typically a d-dimensional hyperbox or hyperball of radius r. In the approximate nearest neighbor (ANN) search, I find the approximate closest neighbor to each point in Q from S. Each answer is approximately correct with high probability. There is always a small chance that another point (the true solution) is even closer. See Mount's paper (Mount and Arya, 2010) for more details on ANN searches.

7.2 Related Work

In this section, I briefly discuss NN solutions, *k*d-trees, and related NN work on the GPU.

7.2.1 NN Solutions

A brute force QNN search could directly compare the query point to all n points in the search set (see Figure 7.1). Solving the All-NN problem this way takes quadratic $O(n^2)$ time.

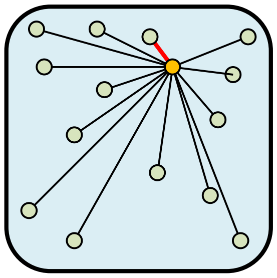


Figure 7.1: Brute Force Spatial Search. The query point is connected by a red line to its nearest neighbor

However, one can achieve better asymptotic performance using spatial data structures, such as fixed grids, Quad-trees, BSP Trees, R-Trees, Voronoi Diagrams, and *k*d-trees (see Figure 7.2). Most of these structures subdivide the original space containing all the points into smaller spatial regions, called cells, and partition the original points into these cells. Many also impose a spatial hierarchy on the cells. NN searches on these data structures use "branch and bound," which focuses the effort on the small set of nearby cells that are likely to contain neighbors and trims away large groups of cells that are too distant. The box tree of Vaidya (Vaidya, 1989) is an early example of a data structure to solve the All-NN problem in $O(n \log n)$ time. For a review of spatial data structures, refer to Samet's book (Samet, 2006).

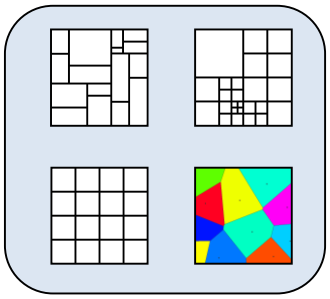


Figure 7.2: Spatial Searches

For my NN search solutions, I focus on the *k*d-tree, a generalized binary tree invented by Bentley (Bentley, 1975) and improved by many researchers in the years since. Arya (Arya and Mount, 1993) detailed an efficient nearest neighbor (NN) algorithm using a depth-first search (DFS) balanced *k*d-tree, a priority queue, and a trim optimization to avoid unproductive search paths. This approach resulted in $O(\log n)$ expected search times for each query point on well distributed point sets. Jensen (Jensen, 2001) implemented a fast and efficient version of Arya's

NN search in his book on photon mapping. Jensen's implementation is the basis for my own CPU *k*d-tree algorithm and has informed the development of my GPU NN algorithms.

7.2.2 *k*d-tree Review

The *k*d-tree is a hierarchical spatial partitioning data structure that is used to organize objects in *d*-dimensional space (see Figure 7.3). The *k*d-tree partitions points and more complicated objects into axis-aligned cells called nodes. For each internal node of the tree, I pick an axis and split value to form a cutting plane. This cutting plane partitions all points at each parent node into left and right child nodes. One or more points contained in the cutting plane may be stored at each node. Variations on *k*d-trees differ in how the cutting plane is picked. Splitting heuristics include median split, empty space maximization, surface area, and voxel volume.

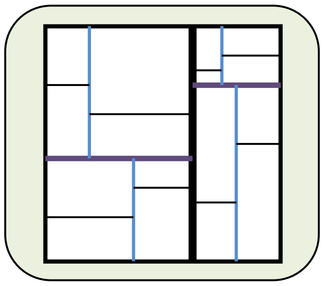


Figure 7.3: A *k*d-tree

A *k*d-tree for a search set *S* of *n* *d*-dimensional points takes $O(d \cdot n)$ storage and can be built in $O(d \cdot n \log n)$ time. I build the *k*d-tree on the CPU and then transfer the *k*d-nodes onto the GPU. (I also implemented a GPU build algorithm, but performance was worse than the CPU build algorithm, so I abandoned it.)

To perform a nearest neighbor search in a *k*d-tree, one can imagine traversing the entire tree, computing the distance of the query to the search points stored at each node while keeping track of the nearest neighbor point found thus far. If the method reaches a node whose bounding box extent is farther from the query point than the current best candidate, that node and all of its children can be skipped. If the method traverses by first visiting the child on the same side of the split as the query point and by visiting the other child only if necessary, then the method will prune many nodes. Search queries (QNN, *k*NN and RNN) that return *t* results have been shown to take worst-case $O(d \cdot n^{(1-1/d)} + t)$ time for all search point sets and expected $O(\log_2(n) + t)$ time for well distributed search point sets. For the 2D All-NN and All-*k*NN searches, I multiply the theoretical cost of a single point query by the number points (n) in my search set, giving $O(n\sqrt{n}+tn)$ worst-case time and $O(n \log_2(n) + tn)$ expected time using a balanced *k*d-tree implementation (Samet, 2006). In addition to performing NN searches, *k*d-trees can also solve point location, range search, and partial key retrieval problems (Skiena, 2008).

7.2.3 Related NN work on the GPU

For GPU solutions, the first NN search solutions were implemented brute force, comparing each of the m points in Q to all n points in S. For each query point q_i, the n query to search point distance calculations are computed in parallel using p threads, which takes $O(n/p)$ time, followed by a parallel reduction to find the minimal distance for that query point, which also takes $O(n/p)$ time. The expected brute-force parallel performance for m query points is $O((mn)/p)$, if we assume the query and search sets are the same size ($m == n$) then this becomes $O(n^2/p)$ time. Purcel (Purcell et al, 2003) approximated a NN search for photon gathering using a multi-pass algorithm involving a uniform grid and an incrementally growing radius. Bustos (Bustos et al, 2006) stored data as textures and used three fragment programs to compute Manhattan distances. He then minimized those distances by reduction. Rozen (Rozen et al, 2008) implemented a bucket sort to partition 3D points into fixed-size grid cells, and he then searched brute force in the 3×3×3 cell neighborhood of each query point. Garcia (Garcia et al, 2008) implemented a brute force NN algorithm in CUDA with a 100+ to 1 speedup compared to the equivalent algorithm in MATLAB. All these authors mention that Arya's *k*d-tree approach is more efficient but is difficult to implement on the GPU due to hardware and software limits.

Zhou (Zhou et al, 2008) built a breadth-first search GPU *k*d-tree in CUDA with a splitting metric that combines empty space splitting and median splitting. This splitting metric approximates either the surface area heuristic (SAH) or the voxel volume heuristic (VVH). The SAH *k*d-tree accelerated ray-tracing, while the VVH *k*d-tree accelerated NN searches. The VVH NN search was iterated using a range region search and by increasing the fixed radius of the search region on each iteration. The GPU *k*d-tree built about 9-13 times faster than the CPU *k*d-tree. The GPU *k*NN search ran 7-10 times faster than the CPU *k*NN search.

Qiu (Qiu et al, 2008) developed a GPU ANN search based on Arya's approach with a *k*d-tree to assist in solving a 3D registration problem on the GPU. The *k*d-tree is built on the host CPU and then transferred to the GPU before running ANN. The ANN search back-tracks to candidate nodes using a small fixed length queue. If the queue is full, new candidate nodes are discarded. Thus final query results are approximate. According to Qui, GPU registration was 88 times faster than CPU registration. Unfortunately, the performance comparison between the GPU and CPU ANN searches was not broken out from the overall results.

7.3 The *k*d-tree Data Structure

My NN search algorithm is adapted from Arya's (Arya and Mount, 1993). It uses a minimal *k*d-tree, a search stack, and the trim optimization. I demonstrate this solution for 2D points, although later in the paper, I also do performance experiments on 3D and 4D points.

7.3.1 *k*d-tree Search Concepts

To help the reader understand the *k*d-tree search, I briefly enumerate the following six concepts:

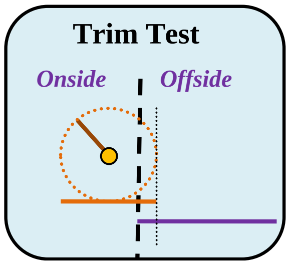


Figure 7.4: *k*d-tree Trim Test

(1) Each *k*d-node contains a *search point* ‹*x*, *y*, ...›.
(2) A best distance variable tracks the closest solution found so far.
(3) A 1D interval trim test eliminates non-overlapping sub-trees (see Figure 7.5).
(4) At any level of my search path, the onside node is the left or right child containing the query point and the offside node is the remaining node.
(5) A depth first search (DFS) first explores onside nodes while storing overlapping offside nodes in a search stack to revisit later.
(6) Each element stored on the search stack contains a *k*d-node index, onside/offside status, split axis, and split value.

7.3.2 *k*d-tree NN Search

My *k*d-tree search algorithm works as follows (see Figure 7.5). The root search element (root index, *onside*, x-axis, ∞) is pushed onto the stack. While the stack is not empty, the top search element is popped off the stack, and the current node index, onside/offside status, split axis and split value are extracted from it. If the node is marked as offside, a trim test is applied in order to accept or reject the entire offside sub-tree. The trim test (illustrated in figures 7.5 and 7.6 is a 1D interval overlap test of a ball with the offside half-plane, where the ball is centered at the query point with radius equal to the best distance and the half-plane is defined by the current split axis and split value. If the node is onside (or offside and accepted), the current *k*d-node is loaded from the node index. Next, if the distance between the query point and the current node's search point is smaller than the current best, my method updates the best distance and best index. The current nodes split axis and value are used to form left and right 1D intervals. The interval containing the query point is the onside node; and the remaining interval is the offside node. The trim test is applied to the offside node in order to keep or reject it. If kept, an offside search element is pushed onto the search stack. The onside search element is always pushed onto the search stack. When the search stack becomes empty, the best distance and best index indicate the nearest neighbor.

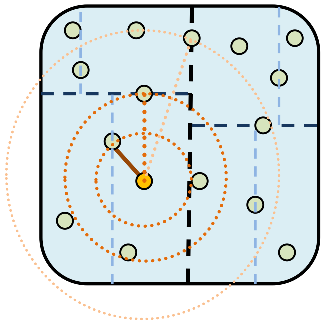


Figure 7.5: *k*d-tree Search

7.4 Hardware Limits and Design Choices

7.4.1 GPU Hardware Considerations

In this section, I discuss GPU hardware limitations and the engineering decisions made in response to these limitations. All my NN kernels were implemented using the CUDA 2.3 API and tested on the NVIDIA GTX 285 GPU. Some of these limitations include the following seven concepts – floating point data, memory hierarchy access speeds, memory hierarchy capacities, memory alignment, coalescence, thread block size, and divergent branching.

Floats: Modern GPUs support both 32-bit and 64-bit floating point data. I focus only on 32-bit floating point data since 64-bit doubles take twice the space and are a factor of 8 slower on the GTX 285 GPU[1]. Floating point support on the GTX 285 is not fully IEEE 754 compliant, and, in a handful of queries, my GPU and CPU NN searches returned slightly different neighbors. In all cases that I investigated, the resulting nearest neighbor distances turned out to be identical. Therefore, the different results were all valid solutions.

Memory Hierarchy: Recall from Chapter 3.4 that the GPU memory hierarchy access speeds, from fastest to slowest, are registers, shared memory, constant memory, and global memory (RAM). For better performance, I put local variables in registers, simple indexed data structures in shared memory, and keep points and *k*d-tree nodes in global memory. Note: I tried storing indexed structures directly in registers; however, CUDA stored them in local memory (AKA a special partition in global memory) instead, which slowed the overall search time by a factor of three.

I minimized the number of data transfers from slower RAM into faster shared memory. The NN search code contains a single read per loop. The total reads per query is $O(\log n)$ expected or $O(d \cdot n^{(1-1/d)})$ worst case. For example, in a QNN search containing 1 million search and query points, each query visits about 40-80 *k*d-nodes (one read per node) to find the exact answer.

Memory Capacity: Since the GTX 285 has only one GB of fixed memory limiting data storage; I sought to minimize the size of my data structures in GPU memory. I compressed *k*d-nodes from eight down to two fields for 2D points. So the 2D QNN search needs only seven

[1] The ratio of double precision to single-precision performance varies across the different Fermi and Kepler GPU cards. For instance, on the GTX 580 double precision (64-bit) is 8× slower than single-precision (32-bit); On the GTX 680, double precision is 24× slower; and on the GTX Titan, double precision is 3× slower. Ideally, double precision should be only 2× slower than single precision.

32-bit elements per 2D point to store query points, *k*d-nodes, and final search results. This approach allowed the QNN search to process up to 36+ million 2D points on the GTX 285.

Memory Alignment: Data structures aligned on 4, 8, or 16 byte memory boundaries perform faster than unaligned data. I saw a 37% speed improvement by aligning my data structures[2].

Coalescence: The GPU coalesces memory accesses only if they are sequential. With a NN depth first search (DFS) through the *k*d-tree, all threads within a thread warp start their searches at the root node but then quickly diverge to different unpredictable sub-trees (based on their individual query point) within the *k*d-tree and thus different parts of memory. NN searches on a DFS *k*d-tree tend not to result in sequential reads across the data warp and thus do not support coalescence. Consequently, I ignore this GPU hardware property at this time.

Latency: The GPU programmer hides latency via TLP by scheduling a large grid of thread blocks; however block performance is limited by the slowest thread in each block (or thread warp). Both grids and thread blocks can be 1D or 2D in shape. A 1D or 2D thread block shape has little effect on performance. So, I excluded my 2D thread block results. The grid can support a maximum of 65,535 thread blocks[3] in any dimension. Each thread block can contain a maximum of 512[4] threads. For the GTX 285, the thread manager maps thread blocks onto 15 SMs each containing 16 SPs. I setup my NN searches to use one thread per query point. I use *padded access* (as described in Chapter 5.1.2) to pad my queries up to the next multiple of the thread block size by repeating the first query as needed; this approach avoids a range check comparison that would increase divergence across the threads within the thread warp.

[2] I aligned my various *k*d-tree search data structures using the CUDA `__align__` macro. See the CUDA programmers guide for more details.
[3] This maximum of 2^{16}-1 (65,535) threads per grid in any dimension is a hard constraint of both the Telsa and Fermi architectures, the Kepler architecture raises this maximum limit to 2^{32}-1 threads per dimension.
[4] 512 threads is the maximum number of threads per thread block allowed on Telsa architectures (including the GTX 285 card). Both Fermi and Kepler architectures increase this limit to 1,024 threads per thread block.

Thread Block Size (TBS): Each GPU core is limited to 16 KB of shared memory and 8 K of 32-bit registers. My current NN searches require about 24-32 registers for temporary variables, which limits the maximum number of threads per SM to 256[5], at most. The QNN and All-NN searches require 192–240 bytes of shared memory for data structures including a 20–28 element deep stack. This shared memory constraint limits the maximum number of threads per SM to at most 64–80[6]. My performance experiments revealed that the optimal thread block size for my DFS *k*d-tree NN searches is 4–16 threads per-block, depending on the search type and the size of the input data.

Divergent Branching: On the GPU, branch divergence degrades performance (see Chapter 3.3). If at least two threads in a thread block diverge at a conditional branch, then both the "if" and "else" branch paths must be executed by all threads in the thread block. So, I eliminated as many branches as possible from my code. The remaining conditional logic is necessary for correct behavior, for which I accept the performance hit due to divergence. I process the All-NN and All-*k*NN searches in sequential *k*d-tree order to increase the coherence of all threads in the thread block. This results in a modest 4–5% performance improvement for the All-NN search over the QNN search. However, the trade-off is that the All-*k*NN search performs slightly worse than the *k*NN search.

7.4.2 *k*d-tree Design Choices

Based on the GPU hardware limits, I sought to efficiently use GPU memory resources. Such a goal suggests bounding the *k*d-tree height and reducing the size of data structures in memory.

[5] Since the Fermi and Kepler architectures increase the register pool size to 32 K and 64 K respectively, the maximum number of threads per SM on these GPU architectures would be raised to 1,024 and 2,048 threads respectively.
[6] Since both the Fermi and Kepler architectures increase the shared memory pool size to 48 KB, the maximum number of threads per SM on both these GPU architectures would be raised to 208-256 threads.

Bounding *k*d-tree Height: Shared memory is the target for my NN search stack. There is only 16KB of shared memory across all threads in each GPU core on the GTX 285. If I use 64 threads per-thread-block, then I have at most 256 bytes available for all data structures. Since my data structures must include a DFS search stack for performing the NN searches, this shared memory constraint bounds the stack size to 20-28 elements, at most. Since my stack size is bounded, I also must bound the length of any *k*d-tree search path to prevent overflowing the stack. One way to prevent this overflow is to bound the maximum height of the *k*d-tree search tree. Bounding the height implies that the *k*d-tree should be both balanced and static. For faster index calculation, I store the *k*d-tree in an efficient array layout. I describe the concepts of balanced, static, and efficient array layout as follows:

> **Balanced *k*d-tree:** A balanced *k*d-tree of maximum height $\lceil O(\log_2 n) \rceil$, with a difference of at most one level across all leaf nodes, is built by setting the cutting plane through the median point of each sub-tree.
>
> **Static *k*d-tree:** A dynamic *k*d-tree can handle insertions, deletions, and modifications, but it can quickly become unbalanced, and exceed my bound on height. When all points are known *a priori*, I can build the *k*d-tree all at once and never change it. A static tree also enables a left-balanced binary tree layout order for the *k*d-tree that supports fast index operations.
>
> **Array Layout:** For faster indexing, I store the *k*d-nodes in an array as a left-balanced binary tree. *k*d-nodes are stored in the range [1..*n*] using one-based indexing. The root is always stored at index one. Given a node at index *i*, its parent is found at $\lfloor i/2 \rfloor$, and its left and right children are found at 2*i* and 2*i*+1 respectively. Any child indices greater than *n* are invalid. Leaf nodes have both invalid left and right child indices. The *k*d-nodes are first built as a left-balanced median *k*d-tree and then converted into a left balanced

binary tree as part of the build process. The left-balanced median position (*LBMpos*) for splitting a partition containing n nodes can be found in 3 steps as follows:

(1) $h = \lceil \log_2(n+1) \rceil$
(2) $half = 2^{(h-2)}$
(3) $LBMpos = half + \min(half, n-2*half+1)$.

Basis Step: The above formulas do not work correctly for small partitions ($n \leq 3$), so the correct *LBMPos* for n=1, 2, 3 are 1, 2, 2 respectively.

Reducing Memory Foot-print: To maximize the number of points in GPU RAM memory, I minimize the size of the *k*d-tree data structure. A maximal set of *k*d-node fields might include: child pointers, parent pointer, split axis, split value, cell bounding box, and stored point.

***d*-Dimensionality:** I use points with 2-4 dimensions ‹*x*, *y*, ...› in these NN searches, which reduces the data stored on the GPU. The searches can be extended to higher dimensions as well; however, since *k*d-tree worst-case performance is no better than a brute-force search for higher dimensions, I recommend not using *k*d-tree based NN searches for points with high dimensionality ($d > 16$).

Eliminating fields: The parent pointer can be avoided by using the search stack in the NN search to back-track. The split axis can be implicit in a cyclic *k*d-tree if that tree splits ‹*x*, *y*, *x*, *y*, ...› and the split value is implied by the stored *d*-dimensional point. Cell bounding boxes are not needed for NN search. Child pointers can be eliminated by computing them directly from the left binary tree layout ($2i$, $2i$+1). Such a computation results in a fully minimal *k*d-tree, where each *k*d-node contains just the original points rearranged into a left-balanced binary tree order. Note: I also need to store a remapping array of size $O(n)$ for converting rearranged node indices back into the original point indices to obtain the final search results.

Final Design: I implement my DFS *k*d-tree data structure as a 2D, 3D, or 4D static balanced cyclical *k*d-tree with a single left-balanced median point stored at each node (internal or leaf). The nodes of the *k*d-tree are stored as a left-balanced binary tree array. The NN search is implemented using a depth first search using a stack for back-tracking. This design bounds the height of the *k*d-tree for predictable stack sizes, minimizes the footprint of the *k*d-tree and search nodes in memory. It also reduces the number of transfers to and from slower global memory.

7.5 Building the *k*d-tree

In this section, I describe how to build the minimal *k*d-tree from the search points. I construct the *k*d-tree on the CPU and then transfer it to the GPU for the GPU NN search. A high-level overview is found in Figure 7.6 (left panel).

```
procedure BuildKDTree(d, points, lbm kd-nodes)
// Initialize kd-tree nodes
n ← |points|
Allocate memory for n median kd-nodes
Allocate memory for n left balanced median
(lbm) kd-nodes
for all in points
  medianNodes[idx].xy ← points[idx].xy
  medianNodes[idx].pointIdx ← idx
  medianNodes[idx].nodeIdx ← INVALID
// Add root build item
top ← 0
build ← { [0,n-1], x-axis, 1 }
buildStack[top++] ← build;
// Build kd-Tree
while buildStack not empty do
  // Get current build item
  currItem ← buildStack[top--]
  [low,high] ← currItem.sequence
  currAxis ← currItem.splitAxis
  currIdx ← currItem.location
  N ← (high-low)+1
  M ← low + LBMpos(N)  // Left bal. Median
  L ← (low+M-1)/2
  R ← (M+1+high)/2
  left  ← 2*currIdx;
  right ← left+1;
  nextAxis = (currAxis+1) % d
  // Partition via Median Selection
  Partition(medianNodes, M) into sub-seqs.
    Left{low,M-1}, Median{M}, and
    Right{M+1,high}
  mNode ← medianNodes[M]
  lbmNode[currIdx] ←
    {mNode.xy,mNode.pointIdx,M}
  // Add right build item to stack
  rightItem ← {[low+M+1,high],nextAxis,right}
  buildStack[top++] ← rightItem
  // Add left build item to stack
  leftItem ← {[low,low+M-1],nextAxis,left}
  buildStack[top++] ← rightItem
end while
// Cleanup
Free memory associated with median kd-nodes
return lbm kd-nodes
```

```
procedure QNNsearch(d, qp, kd-nodes, remap)
// Initialize search
root ← kd-nodes[1]
bestIdx ← 1
bestDist ← Huge Value (Infinity)
// Add root search element
top ←0
rootElem ← {1, onside, x-axis, Infinity}
searchStack[top++] = rootElem;
// Find Nearest Neighbor
while searchStack not empty do
  currElem ← searchStack[top--];
  if currElem.state == offside,
    result ← trimtest(bestDist,
             currElem.splitValue )
    if result == rejected
      return to top of loop // while (search)
    end if
  end if
  // Update Best Distance
  currNode ← kd-nodes[currElem.nodeIdx]
  left ← 2 * currElem.nodeIdx
  right ← left+1
  currDist ← distance( qp.xy, currNode.xy )
  if currDist < bestDist
      bestDist ← currDist
      bestIdx ← nodeIdx
  end if
  currAxis ← currElem.splitAxis
  nextAxis ← (currAxis + 1) % d
  splitValue ← currElem.xy[currAxis];
  determine onside and offside nodes
    from qp, splitAxis, splitValue
  // Add offside node
  diff2 ← (qp.xy[currAxis] -
          currNode.xy[currAxis])^2
  result ← trimtest( bestDist, diff2 )
  if result == accepted
    offElem←{offIdx,offside,nextAxis,splitValue}
    searchStack[top++] ← offElem
  end if
  // Add onside node
  onElem ← {onIdx,onside,nextAxis,splitValue}
  searchStack[top++] ← onElem;
end while
bestIdx ← remap[bestIdx] // map node into pntIdx
return bestIdx and bestDist.
```

Figure 7.6: *Build & Search* Methods
The kd-tree build algorithm from a list of search points (left panel). And the kd-tree search algorithm from the point of view of a single query point (right panel).

As shown in Figure 7.6 (left panel), I compute the minimum and maximum bounds of the search points. The root of the *k*d-tree is conceptually associated with these min-max bounds and the sequence [1, *n*] of original points. A split value is picked along one of the dimensional axes. The split value is chosen to optimize the results according to some measure. All points are partitioned into the two smaller left and right boxes based on the splitting value. Each child node is associated with its bounding box and partitioned sequence of points. The *k*d-tree is recursively refined by splitting each child sub-tree, the associated boxes, as well as associated point sequences until some stopping criteria is reached. In general, there are many possible ways in which the splitting plane can be chosen. For my *k*d-tree, I always choose the left-balanced median point as the splitting plane along the current cyclical axis ‹*x*, *y*, *x*, *y*, ...›, as the tree is descended. The left-balanced median point is found using the `quickmedian` selection algorithm (Sedgewick, 1998). Recursion is converted into iteration by means of a build stack for tracking work yet to be done.

`Quickmedian` Selection Algorithm:

The `quickmedian` algorithm used for selection is similar to the `quicksort` algorithm used for sorting and uses the same `partition` sub-routine. Each selection iteration runs in two phases: pivoting and partitioning.

Pivot phase: The algorithm picks a candidate pivot value *p* using the median of three technique.

Partition Phase: The pivot value is then used to partition the points into three data sets {*Left*: points less than *p*, *Middle*: all points equal to the pivot value[7], and *Right*: all points greater than or equal to *p*.}. If the true median position is equal to the current pivot position, the algorithm stops and returns the pivot point as the median. Otherwise, the algorithm iterates into the child data set (left or right) which contains the true median position. This approach takes quadratic $O(n^2)$ time in the worst case but its expected performance is linear $O(n)$ (Sedgewick, 1998). In practice, this approach is fast and reliable.

[7] Though not required, if desired, points that are equal to the pivot value can then be compared on a secondary key such as index offset or pointer address to preserve *stability* (meaning that if the point *a* precedes point *b* in the array and both *a* and *b* match the pivot point *c* then after partitioning *a* will still precede *b* in the middle partition, in other words their relative orders in the original array are preserved).

7.6 Searching the *k*d-tree

All my NN search solutions are based on the *k*d-tree search solution, already described in Section 7.3. This same search solution can be simplified and adapted to solve the Point Location problem as described in section 7.6.1. I give more details on a CPU Host function for the QNN and All-NN search solutions in section 7.6.2. The *k*NN and All-*k*NN search solutions must track the *k* closest points, so I introduce more details on how to handle these *k* points in 7.6.3. The data structure used to track the *k* closest points also uses shared memory, I talk about how this impacts shared memory resource usage in 7.6.4.

7.6.1 Point Location Problem:

I can quickly find items in a *k*d-tree by traversing down the tree until the cell of interest is found and then by searching for the point of interest within that cell. This approach is easily implemented using the search algorithm described previously in Figure 7.7 (right panel) by simply eliminating the stack and back-tracking code. While traversing the *k*d-tree, the code always chooses the onside node (left or right sub-tree) whose 1D interval contains the query point until arriving at the leaf node which locates the query point.

NN Search Remapping Issue: To solve any NN search problem such as QNN, I directly convert the *k*d-tree NN search algorithm into code. However, one minor change is also required. Since the NN search algorithm actually returns the index of the nearest *k*d-node from the constructed median array layout, but, what is needed is the index of the nearest point from the original search set, then I solve this problem by creating an additional remap array of original indices stored in median array order and then using the median index as a lookup into the remap array to get the original search index. This is a simple straight forward solution but adds another level of indirection and another non-coalesced I/O to the total cost of performing NN searches on GPUs.

7.6.2 QNN and All-NN Search Algorithms

A brief high level overview of the QNN search algorithm is presented in this section. Both the GPU and CPU code are implemented using this algorithm for a fair comparison, see Figure 7.7 (right panel) for more details. For the CPU code, a CPU NN search algorithm is called for each query point in turn. For the GPU kernel, a CPU host function does the following:

(1) Allocates host and device memory for the search points, query points, *k*d-tree nodes, and final results
(2) Sets up the thread blocks for the GPU
(3) Transfers the inputs onto the GPU
(4) Invokes the parallel GPU NN kernel
(5) Transfers the output results back onto the CPU

7.6.3 *k*NN and All-*k*NN Search

The *k*NN and All-*k*NN searches are based on the QNN and All-NN searches. Two simple changes enable these searches. First, the *k* neighbors are tracked by a closest heap data structure and, secondly, the trim distance test is changed to work with *k* points instead of a single closest point.

Closest Heap Data Structure: The *k* nearest neighbors are stored in a closest heap data structure (Jensen, 2001), which acts first like an array and then later like a heap. *Array Behavior*: While visiting the first *k*-1 search nodes in the *k*d-tree, these nodes are appended to the end of the array. Each append takes constant *O*(1) time. *Heap Behavior*: After adding the *k*th search node into the array, the array is converted into a max-distance heap (Sedgewick, 1998) using the `make-heap` method, which takes *O*(*k*) time to run. From then on the NN search treats the closest heap data structure as a true heap. For each subsequent node in the *k*d-tree that is encountered during the NN search, the corresponding search point is compared to the top element of the closest heap. If the distance from the new point to the query point is less than the distance on top of the heap, the top of the heap is replaced with the new point. The correct heap ordering is then restored via the `demote` method, which takes logarithmic *O*(log *k*) worst case time to run.

Unfortunately, the closest heap data structure results in many more instructions and more opportunities for branch divergence across the threads within each thread warp thus slowing down performance (as compared to the QNN and All-NN searches).

During the processing of a NN search, a logarithmic $O(\log n)$ number of nodes will be visited. The worst case time to process the NN search using the closest heap data structure is the time to append the first k points. The time to run `make-heap` when the closest heap becomes full is $O(k)$, and the time to compare and insert the last $\log_2(n) - k$ nodes into the closest heap which takes $O(\log k)$ time per `demote`. Summing these different operations, I arrive at a total of $k \cdot O(1) + O(k) + [\log_2 n - k] \cdot O(\log k)$ worst case time, for which the linear term $O(k)$ dominates. Each individual kNN search thus takes $O(\log n + k)$ expected time. Given p cores, the 2D kNN and All-kNN search algorithms should take $O((m \log n + mk)/p)$ and $O((n \log n + nk)/p)$ expected time respectively for well distributed point sets.

Adjusting Trim Distance: The current trim test must also be adjusted to account for the k nearest neighbors: The initial huge or infinite starting best distance value is not allowed to change for the first k-1 insertions into the closest point heap data structure. After the kth insertion, the trim distance is changed to match the maximum best distance from the top of the closest heap data structure in constant $O(1)$ time. Any time the algorithm decides to change the top of the heap (and `demote`) then the current best trim distance must also change to match the resulting new maximum best distance. This has the effect of making the kNN and All-kNN searches run slower (than the QNN and All-NN searches) due to a slower ramp-up time, as they must completely fill the closest heap with k search nodes before then being able to successfully trim away portions of the search tree using the trim test.

7.6.4 GPU Resource constraints for *k*NN and All-*k*NN search

The same two memory constraints apply with the kNN and All-kNN searches: global memory and shared memory. First, I need to save the k search results for each of the n query points. This means the final result structure takes $O(kn)$ space. There is a maximum of 1 GB of RAM memory on the GTX 285 card. This extra constraint on memory limits the searches to about n = one million and $k = 32$ in the worst case. Another possible configuration is $n = 10$ million and $k = 8$. Second, in order to support kNN on the GPU, I need to carve out space for the closest heap data structure from the same shared memory that I use for the search stack. I assume k will never end up being larger than a maximum stack size of 32 elements. This implies that the kNN and All-kNN searches can be done using about half as many threads as I used for the singleton QNN and All-NN searches.

7.7 Performance Results

In this section, I compare parallel NN search performance on the GPU to serial NN search performance on the CPU. All performance tests were conducted on a GTX 285 (Tesla) using a desktop computer configured as below (see Table 7.2).

CPU Hardware: CPU = i7-920@2.67 GHz, RAM=12 GB
GPU Hardware: 2× *GTX 285* (30 SMs, 240 total SPs, 1.0 GB RAM, 159.0 GB/s peak throughput)
Software: GPU API = CUDA 2.3, C++, IDE = VS 2008, OS = Windows 7, SP1, Pointers = 64-bit
Data: Input size, $n = [10^0 – 10^7]$, in increasing powers of ten.

Table 7.2: *NN Search Experiment Environment*

As part of these performance experiments: I show the cost to build the kd-tree on the CPU, see Section 7.71; TLP experiments to find the best thread block size (TBS) for each NN search type, see section 7.7.2; and experiments to show the resulting performance for increasing input sizes (n) and increasing search sizes (k), see section 7.7.3.

7.7.1 Building the *k*d-tree on CPU

Table 7.3 shows the CPU cost of building the *k*d-tree for different numbers of 2D points.

***n*, # of points**	**1**	**10**	**10^2**	**10^3**	**10^4**	**10^5**	**10^6**	**10^7**
Build Time (in ms)	0.019	0.045	0.151	2.43	22.74	192.52	2,165.31	24,491.28
Time/Pnt (ms/pnt)	.014	.0043	.00165	.00165	.00214	.00163	.00179	.00202

Table 7.3: CPU Build Performance

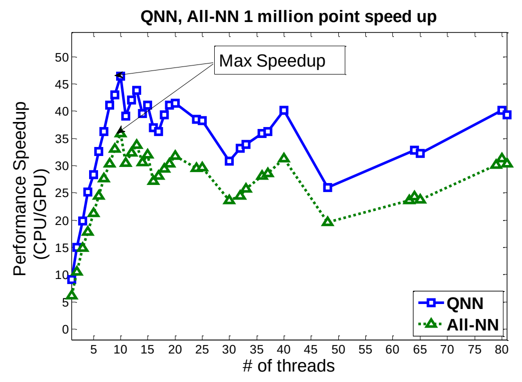
As the previous figure shows, the amortized time per point to build the *k*d-tree initially decreases and then surprisingly levels off after 100 points. I expected the time per point to increase, matching the theoretical $O(n \cdot \log n)$ performance. My best guess for this surprising result is that some CPU caching effects came into play.

7.7.2 Finding the optimal thread block size

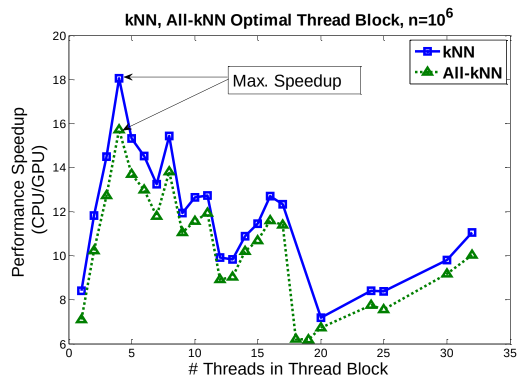
A complicated set of trade-offs determine the optimal number of threads per-thread block. On the one hand, more threads means more parallel work gets done, and therefore there are more opportunities to hide latency stalls, On the other hand, more threads means more competition for resources, increased chances for slower threads to stall the entire thread warp, and more branch divergence.

To find the optimal thread block size on the GTX 285 GPU, I manually tried thread blocks sizes containing between 1 and 80 threads for each of my NN searches. Shown in Figure 7.7 (panels a-c) are 2D QNN, All-NN, *k*NN, All-*k*NN results for data sets containing 1 million and 10 million search points, respectively. For QNN of 1 million search points, the optimal thread block was 10x1 with a speedup of 46.4. For 10 million points, it was 7x1 with a speedup of 43.6. For All-NN of 1 million points the optimal thread block was 10x1 with a speedup of 35.9. For 10 million points, it was 10x1 with a speedup of 36.8. For *k*NN using 1 million search points and k = 32, the optimal thread block was 4x1 with a speedup of 18.1. For All-*k*NN search using 1 million points and k = 32, the optimal thread block was 4x1 with a speedup of 15.7.

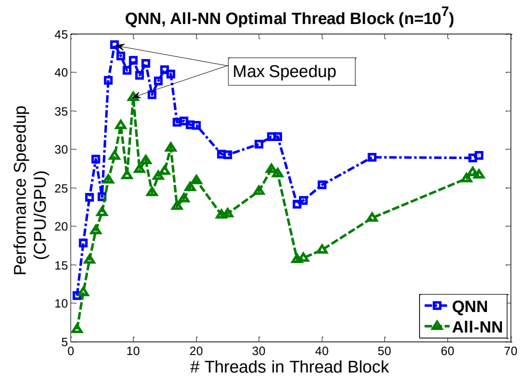
a) **2D QNN**, All-NN searches ($\boldsymbol{n}$ = 10^6 points)



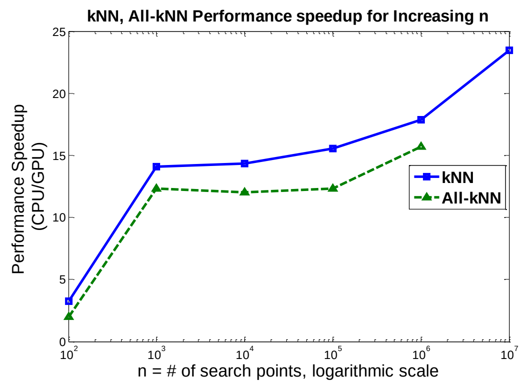
b) **2D kNN**, All-$\boldsymbol{k}$NN searches ($\boldsymbol{n}$ = 10^6 points, $\boldsymbol{k}$=32)



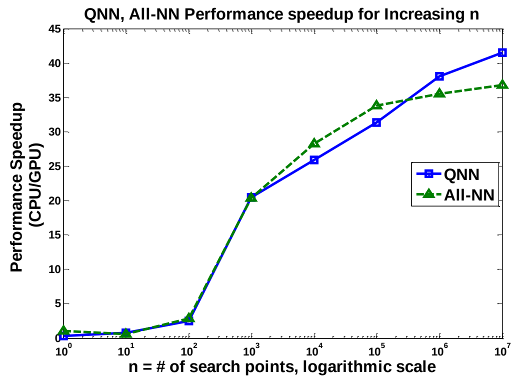
c) **2D QNN,** All-NN searches ($\boldsymbol{n}$ = 10^7 points)



d) **2D $\boldsymbol{k}$NN,** All-$\boldsymbol{k}$NN, Increasing $\boldsymbol{n}$ (TB=4x1, $\boldsymbol{k}$=32)



e) **2D QNN, All-NN,** Increasing $\boldsymbol{n}$ (TB = 10x1)



f) **2D $\boldsymbol{k}$NN, All-$\boldsymbol{k}$NN,** Increasing $\boldsymbol{k}$ (n=10^6,TB=4x1)

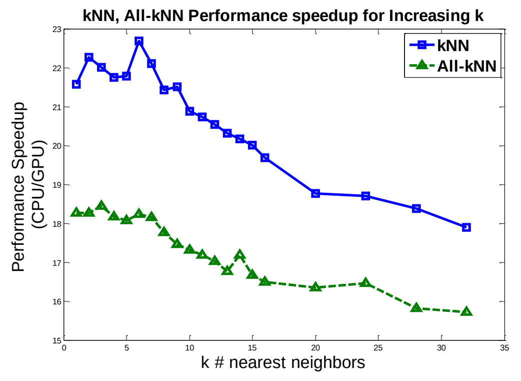


Figure 7.7 ***k*****d-tree Search Results**

a) This chart plots the GPU/CPU speedup for 2D QNN, All-NN searches for increasing thread block sizes with a fixed-size search and query data set of 1 million points. b) This chart is the same but for the 2D *k*NN and All-*k*NN searches. c) This chart plots the 2D QNN, All-NN speedups for 10 million points. d) This chart tracks 2D *k*NN and All-*k*NN speedups for increasing values of *n*. e) This chart tracks 2D QNN and All-NN speedups for increasing values of *n*. f) This chart tracks 2D *k*NN and All-*k*NN speedups for increasing values of *k* from 1-32.

7.7.4 Performance for increasing *n* and *k*

In panels d-f in Figure 7.7, I increased, *n*, the total number of search points across several orders of magnitude using the optimal thread block size for each type of NN search. I keep the number of query points equal to the number of search points in these tests. I plot the four 2D searches in two pairs—QNN and All-NN; *k*NN and All-*k*NN—since they have similar algorithms and results. For the *k*NN searches, I also test performance for increasing values of *k* from 1-32.

***Increasing n*:** For 2D QNN, I see speed-ups in the range [20 - 41.5]; the maximum speed-up occurs for $n = 10$ million. For All-NN, I see similar results: the speedups in the range [20 - 36.8]; the maximum again at 10 million points. For both searches, if $n \leq 100$ points, it is better to use a CPU or brute force solution.

For 2D *k*NN and All-*k*NN, I set $k = 32$. For *k*NN, I see speedups in the range [14 - 18] with the maximum at 1 million points. There is enough memory to run a query with 10 million points, but I then have to decrease $k = 8$ in order for both the search stack and closest heap to fit into shared memory. When I decrease, I see a speedup of 23.4. For All-*k*NN, I see speed-ups in the range [12 - 15.7] with the maximum again at 1 million points. Again, for $n \leq 100$, it is better to use a CPU or a brute force solution.

***Increasing k*:** for the 2D searches, I set $n = 10^6$ and vary *k* from 1 - 32. In both cases, the speed-ups appear to follow a shallow inverse quadratic curve. For the *k*NN search, all the speed-ups are in the range [17.9 - 22.7] with the maximum at $k = 6$. For the All-*k*NN search, the results are similar with speedups in the range [15.7 - 18.4] with the maximum at $k = 3$.

7.8 Conclusion

The demonstrated QNN, *k*NN, All-NN, and All-*k*NN search algorithms are based on a DFS minimal *k*d-tree. The minimal *k*d-tree design is static, balanced, cyclical, using all nodes (internal and leaf) each storing a single point corresponding to the left-balanced median split along the current axis. This *k*d-tree design allows us to handle more points with higher performance by efficient memory utilization. Not only is it possible to support nearest neighbor searches on the GPU using a minimal *k*d-tree but there is a large performance gain from doing so.

7.8.1 2D Performance Summary

The GPU parallel NN searches can handle up to 36+ million 2D points and run faster than the equivalent CPU serial NN search algorithms. The multi-core GPU QNN search runs 20-44 times faster than the equivalent single core CPU search QNN. The GPU All-NN search runs 10-40 times faster than the CPU All-NN search. The GPU *k*NN search runs 13-18 times faster than the CPU *k*NN search. The GPU All-*k*NN search runs 8 - 17 times faster than the CPU ALL-*k*NN search.

7.8.2 3D Performance Summary

The GPU parallel NN searches can handle up to 22+ million 3D points and ran faster than the equivalent CPU serial NN search algorithms. The multi-core GPU QNN search runs 10-30 times faster than the equivalent single core CPU search QNN. The GPU All-NN search runs 10-29 times faster than the CPU All-NN search. The GPU *k*NN search runs 7-16 times faster than the CPU *k*NN search. The GPU All-*k*NN search runs 7 - 14 times faster than the CPU ALL-*k*NN search.

7.2.3 4D Performance Summary

The GPU parallel NN searches can also handle up to 22+ million 4D points and run faster than the equivalent CPU serial NN searches. The multi-core GPU QNN search runs 8-22 times faster than the equivalent single core CPU search QNN. The GPU All-NN search runs 11-21 times faster than the CPU All-NN search. The GPU kNN search runs 6-14 times faster than the CPU kNN search. The GPU All-kNN search runs 6 - 13 times faster than the CPU ALL-kNN search.

7.9 Future Directions

GPU Build: I actually implemented a GPU algorithm to build the *k*d-tree directly on the GPU. Unfortunately, it was actually slower than building the *k*d-tree on the CPU. At some point, I would like to revisit this GPU build algorithm and see if it can be improved.

BFS *k*d-tree: I implemented my DFS minimal *k*d-tree before I understood the GPU hardware and hardware issues that constrain performance—issues such as coalescence, branch divergence, bank conflicts, etc. If I were to re-implement my NN search kernels today, I would try a breadth first search (BFS) *k*d-tree algorithm instead in order to respect coalescence and to help reduce search divergence across threads within each thread warp.

8.0 Case Study: A GPU Histogram

In this case study, I demonstrate my Column data access skeleton (DASk) on a 256-bin histogram over byte data. My histogram implementation is called TRISH[1] and is a deterministic algorithm that avoids atomic operations and gives performance that is data independent. TRISH is higher performing than previous GPU histograms, with a focus on reducing the algorithm's total cycle counts. Reducing the cycles comes from improving thread level parallelism (TLP), instruction level parallelism (ILP) and bit-level parallelism (BLP). TLP improves performance by increasing occupancy from two to three thread blocks, which is achieved by compacting "per-thread" histograms in shared memory and by using register arrays. ILP improves performance by increasing independent instructions via loop unrolling by a factor of $k = [1..63]$ and by batching operations in groups of four. BLP improves performance by compacting bin counts into four 8-bit quads per 32-bit element. BLP also improves performance by reducing binning and by accumulating instructions by working with 32-bit elements as overlapping 16-bit pairs instead of four individual bytes. TRISH runs up to 50% faster than previous GPU histogram methods for random data and 2 to 4× faster for image data.

8.1 Introduction

Histograms, first defined by Karl Pearson (Pearson, 1895), summarize the frequency distribution of a data set. Given an input array V with n data values, with an associated data range $R = [min; max)$ and m bins to count the data values into, the histogram algorithm outputs the histogram as m frequency counts h_i, one for each bin. The actual counting is done using a find

[1] TRISH is an acronym for the "threaded registers independent strided histogram." method

function $f(v)$ that maps each data value into one of sub-ranges[2] $r_i = [a_i, b_i)$, which are known as bins, all sub-ranges r_i collectively cover the original range R without overlap. The m sub-ranges are chosen to be equal-sized as this enables a linear algorithm $O(n)$, as each of n values can be mapped onto its matching sub-range in constant time $O(1)$ using numeric calculations. More complex histograms can support varying size sub-ranges but then require more costly logarithmic $O(\log n)$ or linear $O(n)$ searches in order to map each of n values into its matching sub-range. These more complex histograms thus result in log-linear $O(n \cdot \log n)$ or quadratic $O(n^2)$ asymptotic performance. These more complex histograms will not be discussed further in this dissertation.

The histogram method works in four broad steps:

Step 1) **Subdivide Ranges** partitions the original range R into m equal-sized sub-ranges[3] (known as bins), $r_i = [a_i; a_{i+1})$, all sub-ranges r_i collectively cover the original range R without overlap.

Step 2) **Distribute Values** maps each of n data values from V into one of the m sub-ranges, r_i.

Step 3) **Count Bins** maintains bin counts h_i for the number of data values that fall into each bin r_i.

Step 4) **Output Histogram** outputs the m frequency counts (h_i) that collectively make up the histogram.

The resulting histogram is often graphically rendered as a bar-chart of these counts. The shape of the chart gives information about the frequency distribution of the data.

The histogram methods discussed in this chapter focus on the special case of 256-bin histograms for (8-bit) byte data, which are useful for image processing, text processing, and other applications. The advantage of using 256-bin histograms on byte data is that computing ranges and distributing values into bins can be replaced by simpler indexing operations for faster

[2] There are m subranges $r_i = [a_i, a_{i+1})$ with each of the $(m+1)$ partition values a_i typically computed as $a_i = min + i \cdot [(max\text{-}min)/m)]$.

[3] There are m subranges $r_i = [a_i, a_{i+1})$ with each of the $(m+1)$ partition values a_i typically computed as $a_i = min + i \cdot [(max\text{-}min)/m)]$.

performance. Figure 8.1 shows code for a simple 256-bin CPU histogram method for counting byte values.

```
Input:    V = array of n bytes to be binned
Output:   bins = array of m = 256 bin counts
integer bins[256] = 0;    // Zero counts
foreach idx in [0..n-1]   // Count bytes
  bins[V[idx]]++;
end idx
```

Figure 8.1: A 256-bin CPU histogram method.

Although histogram methods are straightforward to implement on a sequential CPU they have proven difficult to adapt for use on many-core processors such as GPUs. Prior researcher's GPU histogram implementations generate correct results but achieve only 6–15% of the theoretical peak throughput of modern GPU hardware, such as NVIDIA Fermi or Tesla cards. My TRISH method improves modestly on previous results, achieving ~21% of peak throughput.

8.1.1 Parallelism improves Performance

Readers are already aware of the benefits of using instruction-level parallelism (ILP), data-level parallelism, and thread-level parallelism (TLP). In this chapter, I introduce Bit-level parallelism (BLP) which can also improve performance.

Bit-Level Parallelism: Bit-level parallelism (BLP) is an older form of parallelism (Wadleigh and Crawford, 2000) and is a special case of vector-parallelism[4]. BLP processes multiple small data types at once using a single ISA instruction on larger data types. A shown in Figure 8.2, if the native machine word size is 32-bit then the programmer can choose to work with four 8-bit bytes, or two 16-bit words, or one 32-bit double-word (DWORD) using 32-bit ISA instructions.

[4] Bit-Level parallelism is also sometimes referred to as "SIMD within a Word" (SWAR).

Working with 8-bit bytes or 16-bit words on such an architecture using BLP can result in speedups up to 4× or 2× respectively.

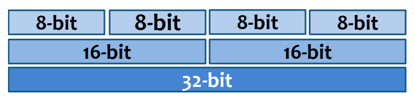


Figure 8.2: *Bit-Level Parallelism*

The trade-off with BLP, is that programmers also pay overhead in the form of extra instructions to compress and decompress multiple small data values into or from the larger data type registers. A performance increase thus depends on whether the instructions saved by BLP is more than the overhead incurred compressing values and decompressing results.

Performance:

To improve parallel performance, I take advantage of these 4 types of parallelism -- ILP, data-level, TLP, and BLP.

To achieve good GPU performance, I use coalescence, avoid bank conflicts, and minimize branch divergence. For measuring GPU performance, I use the concepts of I/O throughput (GB/s) and Total Cycles (TC = II/IPC) from Chapter 3. Using the TC metric, I seek to improve performance by either reducing the total instructions issued (II) or by increasing the instructions retired per cycle (IPC).

To achieve better GPU performance in my TRISH histogram, I also use a couple of ideas from other sources. First, Dr. Micikevicius (Micikevicius, 2010 both papers) explained how to find and remove GPU performance bottlenecks by carefully measuring and analyzing performance. Second, Dr. Merrill (Merrill and Grimshaw, 2010 both papers) used register arrays for better performance and memory utilization by partitioning array elements across the registers of multiple threads in a thread block.

I aim for my GPU histogram methods to be correct, fast, and predictable. I verify correctness by comparison against a baseline CPU method. I achieve faster performance by porting my original CPU methods onto GPUs via data-level parallelism and by seeking to minimize the total cycles that my GPU method takes to complete. I also seek predictable methods where performance is not influenced by the underlying data distribution.

8.2 Related Work

Since my work is related to two prior GPU Histogram implementations, I will briefly describe those two prior GPU histograms from Podlozhnyuk (Section 8.2.1) and Nugteren (Section 8.2.2). There is also a third GPU histogram implementation, by Yang (Yang et al, 2008), but his histogram method is slow (~ 0.7 GB/s throughput on G80 class GPUs).

8.2.1 Podlozhnyuk's Histogram Method

NVidia provides 64-bin and 256-bin histogram methods in the most recent CUDA software development kit (NVIDIA, 2012, C++ Programming Guide). These methods[5] were created by Victor Podlozhnyuk (Podlozhnyuk, 2007). These methods target older GPUs with 16 KB of shared memory per streaming multi-core (SM). Thus, 64-bin histograms can be created in shared memory for each thread. Since 256-bin histograms for each thread would not fit into shared memory, Podlozhnyuk uses a novel "per warp" method, in which thread blocks with six warps of 32 threads each create six histograms in shared memory. After scanning the data, he then accumulates these per-thread or per-warp histograms into the final histogram. Using this "per warp" memory layout, all 32 threads in each warp must compete to increment and update shared histogram bins. In order to maintain correct behavior during collisions between threads, the method must use either hardware atomics or the author's novel software tagging scheme. In this scheme, the top five bits of a counter are reserved for a thread ID (thread offset within the warp), which each thread sets as it tries to increment a bin count. Since hardware guarantees that a single thread will always win despite collisions, each thread must check the winning tag to ensure that its own increment occurred or try again. Of course, this means that worst-case

[5] Shams (Shams and Kennedy, 2007) has generalized Podlozhnyuk's method to support 32-bit inputs and a range of bin sizes.

behavior (many thread collisions) can be 8–32× times slower than best-case behavior (no thread collisions). However, this method uses 192 threads per-block with eight concurrent blocks per SM and thus achieves full occupancy (100% = 1536/1536). Most of this histogram's performance comes from TLP, but performance is limited by the need to resolve intra-warp thread collisions.

Unfortunately, Podlozhnyuk's 256-bin method performs at only 15.0% of peak throughput on modern GPUs (GTX 580) for random, uniformly distributed data. Moreover, performance is data dependent: the data distribution determines the number of thread collisions within each warp, which gates overall performance. As my experiments show, performance can indeed degrade by factors of > 30.

8.2.2 Nugteren's Histogram Method

Nugteren (Nugteren et al, 2007) reported a histogram method for fixed-size images (2048×2048 bytes) that is independent of the data distribution. They claimed that on a GTX 470 they achieved 56% better throughput than Podlozhnyuk's method. Nevertheless, I have not seen consistent improvements in my tests on the more modern GTX 580, 480, and 560M. Two factors may explain this. First, in my tests I replace Podlozhnyuk's older software tagging with newer hardware atomics for a 50% speedup of his algorithm, and, secondly, Podlozhnyuk's data dependent method will perform better on the uniform distribution that forms my main test case than on images with correlated pixel values that were probably used by Nugteren. Our experimental results (as seen in Figure 8.7) are consistent if both factors are taken into account.

Nugteren use the larger shared memory (48 KB) of modern GPUs to implement "per-thread" histograms. Histogram bin counts are compressed by storing two 16-bit bins per 32-bit DWORD. Therefore, 128 DWORDs suffice to store an entire 256-bin histogram. With 32 threads per-thread block, 16 KB of shared memory is also sufficient to store individual histograms for each thread block. This compression enables three thread blocks (of 32 threads each) per SM to run concurrently. Unfortunately, the large amount of shared memory used results in low occupancy (6.25%= 96/1536), and therefore Nugteren's method does not fully utilize the GPU pipelines.

Nugteren's histogram GPU kernel works in a straightforward way. First, the data is partitioned across all the threads in a cooperative thread array (CTA). Next each thread bins and counts all its assigned data. After all input data has been binned, the per-thread histograms are then summed to create a per-block histogram in global memory. A tail GPU kernel is then executed to sum the per-block histograms into the final histogram.

In my experiments, Nugteren's code achieved about 5.4% of peak throughput on the GTX 580 for a fixed image size of 2048×2048 bytes. Performance for this method is predictable, since the method is deterministic and not influenced by the underlying data distribution.

8.3 My TRISH Method

Like Podlozhnyuk's and Nugteren's methods, my GPU histogram method supports coalesced memory accesses for efficient I/O, stores intermediate results in shared memory, and uses simple indexing. My "threaded registers independent strided histogram" (TRISH) method is similar to Nugteren's method with several simple ideas that improve overall performance. Where possible, I report the improvements in throughput (GB/s) afforded by each idea to indicate their contributions to overall performance. I defer the description of the experimental setup to Section 8.4.

For my TRISH Histogram case-study, I use these ideas to achieve better throughput:

- Reduce the total cycles required to complete my method ($TC = II/IPC$)
- Improve pipelining, as measured by instructions retired per cycle (IPC), via a combination of thread- and instruction level parallelism (TLP and ILP)
- Reduce instructions issued (II) by simplifying code and applying vector processing (VP) ideas
- Build compact per-thread histograms in shared memory to achieve Better TLP
- Increase independent instructions by loop unrolling and batching operations into groups of four to achieve better ILP

- Work with 16-bit pairs or 8-bit quads as single 32-bit elements instead of separately as 8-bit bytes to achieve better BLP

Like Nugteren's method, I use a deterministic per-thread histogram algorithm and therefore have no need for tagging or atomics to resolve intra-warp thread collisions as in Podlozhnyuk's method. Thus my method is data independent and faster.

8.3.1 Improving TLP

One way to improve GPU performance is to increase TLP. I increase TLP for my TRISH method by building compact "per-thread" histograms in shared memory. This improves thread-level parallelism at the cost of more frequent accumulation passes.

Compacting Histograms: I compact my TRISH histogram by storing four 8-bit bin counters per 32-bit word. Since each bin counter is a single byte, 64 DWORDs suffice to store an entire histogram for a thread (see Figure 8.3).

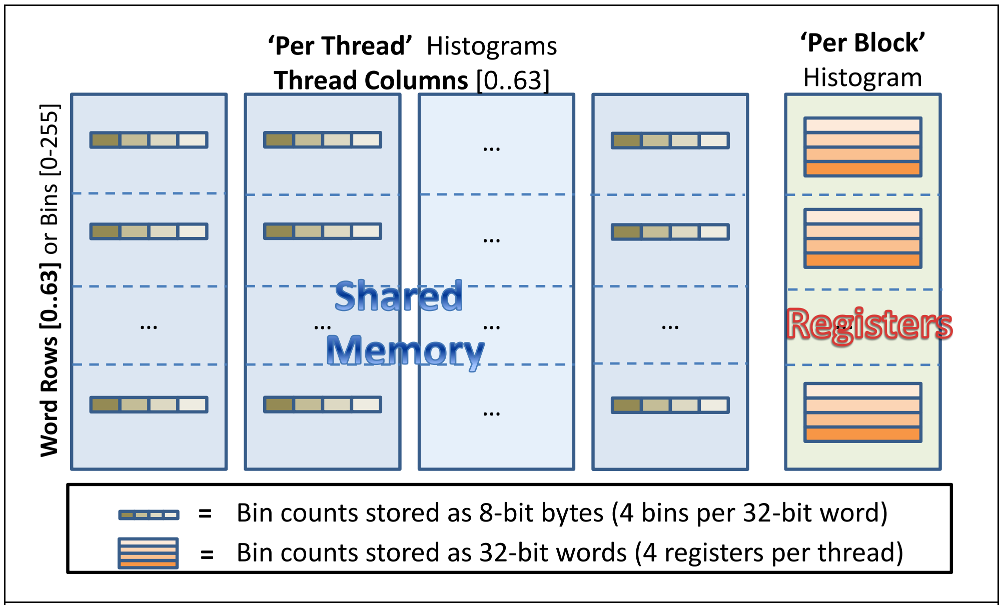

'Per Thread' Histograms
Thread Columns [0..63]
'Per Block' Histogram
Word Rows [0..63] or Bins [0-255]
Shared Memory
Registers
= Bin counts stored as 8-bit bytes (4 bins per 32-bit word)
= Bin counts stored as 32-bit words (4 registers per thread)

Figure 8.3 *TRISH Layout*: – In TRISH, each thread has a 256-bin histogram stored as a column of 64 DWORDs (four 8-bit bin counts per DWORD). Each Thread also maintains four 32-bit registers for the per-block register array.

As can be seen in Figure 8.3, I choose to use 64 threads per-thread block, thus 16 KB of memory suffices to store all per-thread histograms for each thread block. Recall from Chapter 2 that each Fermi and Kepler class SM has 48 KB of on-chip shared memory, this specific data layout in shared memory enables three concurrent thread blocks per SM (3 = 48/16), achieving a thread occupancy rate of 12.5% (192/1536). As a result, TRISH has a much lower occupancy than Podlozhnyuk's method but double the occupancy of Nugteren's method. Furthermore, having two or more warps per SM enables the dual issue feature on Fermi class GPU cards and therefore enables the scheduling of two warps of instructions at a time onto GPU pipelines.

The small range [0..255] of bytes as bin counters means that TRISH must act earlier to prevent overflow. Thus, I accumulate per-thread histograms into a per-block histogram on a regular basis. The per-block histogram uses 32-bit bin counters, and so overflow within the per-block histogram is not a concern. For faster performance via BLP, I treat data elements as 32-bit DWORDs instead of individual 8-bit bytes, meaning each 32-bit DWORD contains four 8-bit data values. To always avoid overflow within the per-thread histograms, I accumulate my per-thread results after binning at most 63 32-bit DWORDs per-thread. That means that I conservatively assume all 252 bytes in those 63 DWORDs will bin into the same counter.

Per-thread histograms are accumulated into the per-block row sum histograms by simply reducing (linear serial scan) the bin counts. As part of the reduction, I reset the per-thread bin counts to zero. Since there are 256 bins but only 64 threads per-thread block, each thread must accumulate four bins.

Direct indexing during the `per-block' row sum accumulation would result in a 32-way bank conflict per warp, which would hurt performance. Consequently, I stagger starting indices by thread ID and index modulo block size (circular indexing). The staggered start ensures that each thread within a thread warp starts on a unique memory bank within shared memory thus avoiding bank conflicts. Circular indexing ensures that the code wraps around correctly to accumulate the 64 DWORDs along each row that will be accumulated into the per-block histogram. Figure 8.4 shows how I implement staggered start and circular indexing.

```
// Staggered Start
currIdx = threadID;
…
// Circular Indexing
nextIdx = mod(currIdx+1, BlockSize);
```

Figure 8.4 *Staggered Start / Circular Indexing*: To help avoid bank conflicts, I use a staggered start along with circular indexing so each thread refers to a different memory bank in shared memory.

Register Arrays: Since each bin counter is completely independent, I can safely store the entire per-block histogram as a register array. I partition the entire 256-bin histogram across all the 64 threads in each thread block. Each thread maintains its four per-block bin counters in registers.

8.3.2 Improving ILP

Since my method results in low occupancy (12.5%), I cannot take full advantage of TLP to keep the hardware pipelines busy. Instead, I fall back on well-known ILP techniques of loop unrolling ,software-pipelining and batching four runs of independent instructions.

***Loop Unrolling*:** I define work per-thread, k, as the number of 32-bit words that each thread counts in bins before looping to the next chunk of work and repeating. To prevent possible overflow in the 8-bit per-thread counters, I should bin and count at most 63 DWORDs (252 bytes) of input values before accumulating my per-thread counts into my per-block counts. Consequently, I pick k in the range [1..63]. Larger values of k amortize the cost of loop overhead across multiple elements and increase the pool of independent instructions that the compiler and the hardware can potentially harvest during instruction pipeline scheduling.

***Software Pipelining*:** To decrease pipeline stalls and improve ILP performance, I group and reorder batches of instructions from k independent work-items. However, unrolling a loop of *k* work-items, could result in up to *k*× as many sets of registers, greatly increasing register pressure on each thread.

***Batching Instructions*:** To decrease register pressure, I batch software pipelining into smaller fixed-size groups of [2-4] work items per batch. In other words, instead of software pipelining all *k* work-items at once, I group the *k* work-items into smaller batches of four work-items at a time (1-4, 5-8, 9-12, …, 57-60, 61-63) and then software pipeline each individual batch of four work-items separately. Batching software pipelining in this manner reduces the registers required from *O*(*k*) down to *O*(*4*). Batching thus provides a good compromise between ILP performance and high register pressure. For GPU programming, I recommend batching a large group of *k* work-items into smaller batches of [2-4] work-items each as a good initial starting point until you have a feel for how many registers are being consumed by your algorithm.

Even with batching, my unrolling and software pipelining of *k* work-items increases register pressure on my TRISH kernels; Using CUDA 4.0, my TRISH method uses 36 registers per-thread (between Podlozhnyuk's 18, and Nugteren's, which spills out of 42 registers.) Since thread occupancy is already limited by shared memory constraints, any reduced occupancy due to high register usage effectively goes unnoticed.

8.3.3 Improving Bit-Level Parallelism

As much as possible I handle the input data and initial count arrays either as 32-bit quads of bytes or as two alternating pairs of bytes. Such an approach reduces the arithmetic operations involved by up to factors of four or two, respectively (Wadleigh and Crawford, 2000). There are two main places where I take advantage of bit-level parallelism (BLP) to improve performance – when accumulating per thread counts into per block counts, and when binning four 8-bit values into their respective bins.

Accumulation Optimizations: One normally thinks of the four 8-bit bins in a 32-bit word as a quad layout [3,2,1,0] as shown in Figure 8.5 (top).

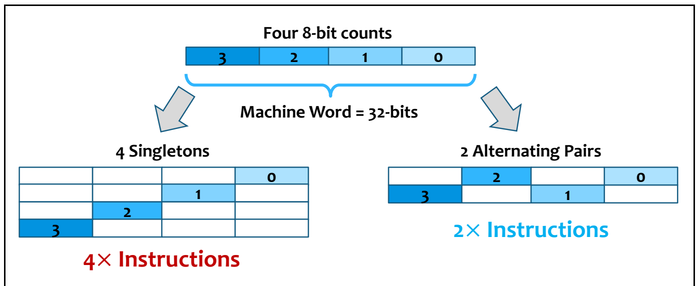


Figure 8.5 *Optimizing via BLP*: Processing 2 or 4 bytes simultaneously using 32-bit instructions.

If we extract each 8-bit value individually as a singleton and then accumulate, then this approach can take up to 4× as many instructions as accumulating the 32-bit DWORD directly as shown in Figure 8.5 (left). Due to overflow issues across bytes within the 32-bit DWORD, it is not always possible to accumulate the 32-bit DWORD directly[6]. However, one can also consider the same 32-bit layout [3,2,1,0] as two alternating pairs [*,2,*,0] and [3,*,1,*] as shown in Figure 8.5 (right). Using this alternate layout, I can extract and work with these alternating pairs using bit shift and mask operations to upscale the arithmetic range from 8-bits to 16-bits before overflow becomes a problem. Changing my accumulation code to work with pairs instead of singletons during the per-block row sum operations improved throughput performance by about 5% on the GTX 580 (from 38.1 GB/s to 40.3 GB/s).

Binning Optimizations: It is common to extract individual bin indices by multiplying the byte value by the thread block size and then adding in the thread offset. However, using bit manipulation, I can multiply alternating pairs of bytes, and, assuming fixed powers of 2 for my histogram table sizes, I can replace expensive multiplies and divides by bit shift and mask operations. This approach not only saves instructions via BLP but also uses less expensive bit

[6] As an alternative, one could mask away the high order bit in each byte before accumulating and then restore them after accumulating. However, the number of overall instructions involved may not change for the better (The number of additions decreases but at the cost of increased mask operations).

operations. I also realized that by using BLP that I could also combine multiple shifts into a single instruction. See Figure 8.6 for an example of my binning code which bins four 8-bit values stored as a single 32-bit DWORD.

```
template < ColSize, nWarps >
Bin4_None(cntPtr, val32, currWarp )
  // Masks (lane and shift)
  const U32 maskRow13 = 0x07E007E0u;
  const U32 maskCol   = 0x03030303u;
  // Get Lane Info [0..3]=bin[0..255] % 4
  laneCol = val32 & maskCol;
  LI_13 = laneRow13 & maskRow13;
  LI_24 = laneRow24 & maskRow13;
  // Get Shifts [0,8,16,24] = [0,1,2,3]*8
  shift = laneCol << 3u;
  // Get Local Indices
  LI_4 = LI_24 >> 16u;
  LI_3 = LI_13 >> 16u;
  LI_2 = LI_24 & 0xFFFFu;
  LI_1 = LI_13 & 0xFFFFu;
  // Get Shifts
  S4 = (shift >> 24u);
  S3 = (shift >> 16u);
  S2 = (shift >>  8u);
  S1 = (shift & 0xFFu);
  S3 = s3 & 0xFFu;
  S2 = s2 & 0xFFu;
  … // Continued on next column

  … // Continued from prior column
  // Get Increments
  inc4 = 1u << S4;
  inc3 = 1u << S3;
  inc2 = 1u << S2;
  inc1 = 1u << S1;
  // Increment bins
  oldCnt = cntPtr[LI_4];
  newCnt = oldCnt + inc4;
  cntPtr[LI_4] = newCnt;
  oldCnt = cntPtr[LI_3];
  newCnt = oldCnt + inc3;
  cntPtr[LI_3] = newCnt;
  oldCnt = cntPtr[LI_2];
  newCnt = oldCnt + inc2;
  cntPtr[LI_2] = newCnt;
  oldCnt = cntPtr[LI_1];
  newCnt = oldCnt + inc1;
  cntPtr[LI_1] = newCnt;
end Bin4_None
```

Figure 8.6 *Binning Code*: This code bins four 8-bit values compacts in a single 32-bit DWORD. The code uses unrolling (group of 4) and interleaving for increased ILP. The code also uses bit masks and shifts tricks for increased BLP.

As shown in Figure 8.6, my binning algorithm depends on knowing the data layout in advance. The data layout is one histogram per thread with each histogram consisting of 64 rows and each row consisting of four 8-bit counters stored as a single 32-bit DWORD. Since there are 64 threads per thread block, there are 64 columns of histograms. Each 8-bit value to be binned is turned into a row [0-63] and shift increment value [0, 8, 16, 24] within each row. Since the data layout is known and fixed, I can use bit tricks via masks and shifts to eliminate multiplies and modulus and use some BLP tricks to reduce total operations. Using the row indices and shift increments, the per-thread histogram bin counts are then incremented, one at a time. Incrementing is by far the slowest part of the binning algorithm due to shared memory accesses and dependencies between instructions within this section of code for each bin operation. I attempted several rewrites to improve performance by interleaving the “increment bins” operations to increase ILP. However, these rewrites failed as they caused incorrect results (two or more work-items colliding on the same bin results in some increments getting lost).

8.3.4 Picking the best k value

What *k*-value (work per-thread) in the range [1..63] gives the best performance? Recall that 63 is the maximum number of elements that can be safely processed before overflow becomes a possibility. Picking the best *k*-value is complicated by several limitations: loop overhead, ILP, and the row-sum efficiency ratio.

Loop Overhead: The TRISH method amortizes the cost of loop overhead across the work of binning k elements per loop. This suggests that TRISH should favor larger values of k.

ILP: At low values of k, the method does not unroll the loop enough to unlock sufficient independent instructions for the warp schedulers to exploit to hide latency during instruction scheduling (see Figure 8.7, for $k = [1..4]$). This suggests that TRISH should favor larger values of k for improved ILP.

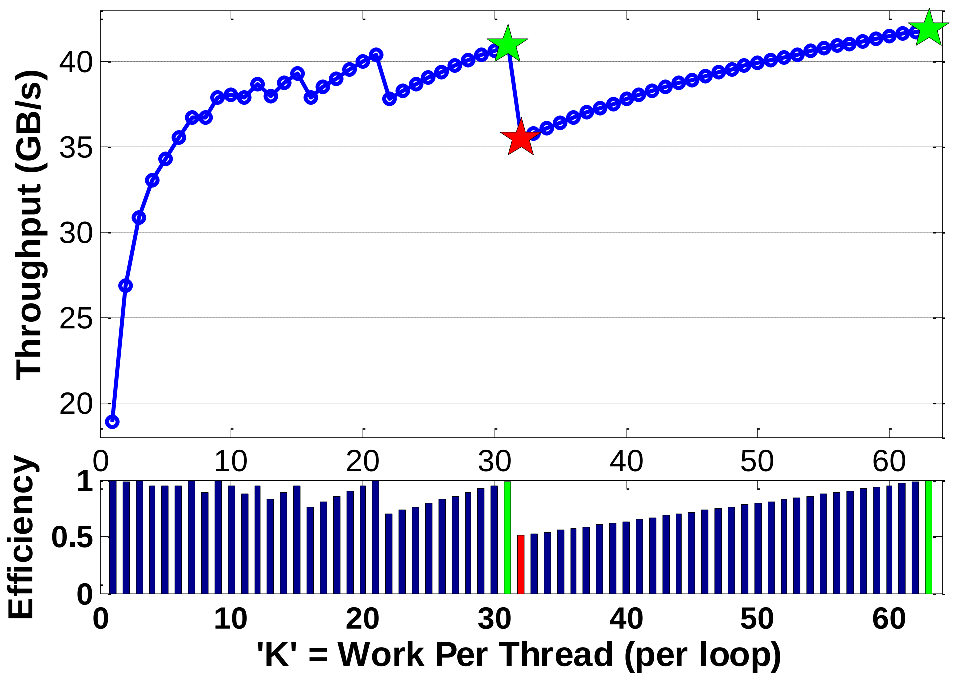


Figure 8.7 ***Row-Sum*** **Efficiency:** ***Upper:*** Impact of work per-thread on I/O throughput for $k \in [1, 63]$. Best results are for k=31 and k=63. ***Lower:*** Bins per row-sum efficiency for $k \in [1, 63]$. Max Efficiency = 1.0 (63 elements binned per row-sum) occurs at k = 1,3,7,9,21, and 63.

Row-Sum **Efficiency:** See Figure 8.7, In order to avoid overflow, TRISH frequently accumulates all per-thread counts into per-block counts and resets the per-thread counts to zero. This accumulation operation, which I call a *row-sum*, is expensive but fortunately is amortized across the cost of binning k DWORD values. Since, 63 DWORDs (252 bytes) is the most the method can safely bin before accumulating, the ideal bins per row-sum efficiency ratio (*bins*/*row-sum*) would be 63 binning operations per row-sum operation. However, since TRISH tests for overflow only at the top of each loop of k work-items, the actual bins per row-sum efficiency ratio becomes the maximum number of iterations before overflow becomes possible, which is computed as *Efficiency* = $(\lfloor 63/k \rfloor k)/63$.

Consider the case, k = 32 (red bar in Figure 8.7), TRISH processes and bins one loop iteration of 32 elements (128 bytes). In the worst case, all 32 elements could end up in one bin counter (bin count = 128). Thinking about what could happen on the next loop, one realizes that overflow is now possible, since (32+32) > 63 or alternately (128+128 > 252). This means that a row-sum operation must be performed on the current iteration to avoid this potential overflow on the next iteration. This results in an efficiency of only 32/63 = 0.508, which turns out to be the worst case bins per row-sum efficiency. Now consider the case $k = 31$ (the green bar in Figure 8.7). With this value, TRISH processes 31 elements (124 bytes) and, on the next loop, can process another 31 elements (124 bytes) before having to perform a row-sum at the top of the third loop, since $(31+31) \leq 63$. This case achieves an efficiency of $(2\times31)/63 = 0.984$. Contrasting these two cases, TRISH must accumulate row-sums almost twice as often for $k = 32$ than for $k = 31$. The ideal bins per row-sum efficiency ratio of 1.0 = (63/63) is consequently achieved at the values $k = 1, 3, 9, 21$, and 63, which are the values of k that divide 63 evenly.

As shown in figure 8.5 (upper graph), I tested all values of $k = [1..63]$ to see how these limiting factors combine. For low values of k ($k \leq 4$), throughput was poor because of high loop overhead and the low potential for increasing ILP. For example, at $k = 1$ throughput was only 18.92 GB/s. At higher values of k, there is a strong correlation between the bins per row-sum efficiency and I/O throughput as the row-sum operation is expensive. I found that $k = 31$ and $k = 63$ give the best throughput results at 40.91 GB/s and 41.86 GB/s, respectively. At $k = 32$, performance drops to 35.44 GB/s due to the increased frequency of row-sum.

Although throughput for $k = 31$ is slightly slower than for $k = 63$, I recommend $k = 31$ as the better overall choice because it is is faster overall for small to medium input sizes (n).

8.3.5 TRISH Method Summary

Here is a brief high-level summary of my 256-bin TRISH code. A CPU host function (not shown) implements my TRISH method by dealing with setup, invoking the GPU kernels, and cleaning up resources.

The actual GPU histogram method is a two-step process using two GPU kernels, a `Count` kernel and a `BlockSum` kernel. I briefly describe the `BlockSum` kernel first as it is so simple. The `BlockSum` kernel simply sums the results of all the individual per-block histograms generated by the `Count` kernel to obtain the final histogram.

The `Count` kernel (see Figure 8.8) implements the Column DASk (see Section 5.3), which groups the data into multiple fixed-size data blocks and partitions the data blocks across a fixed number of columns. Each thread block then cooperatively strides through its assigned data along the column (row by row). A last partially full row must range check to avoid access errors.

Count Kernel

Inputs: $\lfloor n/nBlocks \rfloor + [0,1]$ 32-bit values to bin & count
Outputs: per-block counts ($256 \times nBlocks$)

I/O Summary:
$\lceil n/32 \rceil$ coalesced reads of input values
$\lceil (256 \times nBlocks)/32 \rceil$ coalesced writes of per-block counts

Algorithm: **(In Parallel, 64 threads)**

```
                                            // 1. Set-up
Compute # of 'full' rows & 'left over' partial row
while (more 'full' rows)                    // 2. Main loop
  if (overflowCount ≥ (63 – K))
    row-sum( … )  // Accumulate block counts
    overflowCount = 0;
  end if
  repeat K times  // fixed K in [1..63]
    Read 32-bit DWORD of data
    Bin 32-bit DWORD as 4 bytes
  end repeat
  overflowCount += K;
  move to next data block along column (next row)
end while
if ('left-over' row)                       // 3. Partial Row
   … Similar to "main loop" plus "range checks"
end if
                                           // 4. Wrap-up
if (overflowCount > 0)
  row-sum( … )  // Accumulate block counts
end if
Write out block counts
```

Figure 8.8 *GPU Count Kernel* – Computes per-block histograms for each thread block

As shown in Figure 8.8, each thread block computes its own per-block histogram for its assigned data elements. Following the Column DASk template, this code has four main sections: (Set-up, main loop, partial row, and wrap-up)

> Section 1) **Setup:** The set up code figures out how many fixed size rows (via the stride) to process in the main loop and checks if we have any leftover data that needs to be processed with careful range checks.
>
> Section 2) **Main-Loop:** In the main loop, each thread in a warp (of 32 threads) processes *k* work-items in each data block. For the k^{th} work-item, each thread reads in a 32-DWORD from global memory in a coalesced manner and then bins each of the four bytes within that DWORD into the per-thread histogram array kept in shared memory. An overflow counter is incremented after each inner loop of k elements is processed. On detecting possible overflow, the code launches the `row-sum` operation to accumulate per-thread counts into the per-block histogram stored as a register array, the per-thread counts are reset to zero as part of the row-sum. The `row-sum` function treats 32-bit quads as two alternating pairs for better performance via BLP.
>
> Section 3) **Partial-Row:** The Column DASk pushes any range checking into the last row only, which most likely is only partially full. This section of code is almost exactly the same as the main-loop code with the addition of extra required range-checks when loading data from global memory. Since the last partially full row contains one row of fixed-size data blocks and each data block contains *k* work-items. I actually use a telescoping "divide and conquer" tree on *k* to reduce the actual number of range checks required from $O(k)$ to $O(\log k)$[7] at the cost of more verbose and confusing code.
>
> Section 4) **Wrap-Up:** The wrap up code then accumulates any left-over per-thread counts and writes out the final per-block histograms to global memory in a coalesced manner.

8.4 Performance Results

In this section, I report the performance results of three different sets of experiments. In the first set of experiments, I directly compare my TRISH histogram method to Podlozhynuk's and Nugutern's histogram methods on both synthetic data sets and image data sets. In the second set of experiments, I deliberately turn off some of my optimizations to see the degraded

[7] Telescoping means that I do my range check first on a group of 32 work-items, then 16 work-items, then 8, then 4, then 2, then 1. This requires $\log_2(k)$ range checks per data block instead of *k* range checks per data block (one per work-item).

performance that results. In the third set of experiments, I collect useful performance metrics from some the hardware profile counters on the GPU card.

Hardware	CPU = i7-920 @ 2.66 GHz, RAM = 12 GB, GPU = GTX 580 (16 SMs, 512 SPs, 1.5 GB memory, 192.4 GB/s peak throughput).
Software	GPU API = CUDA 4.0 (64-bit), LANG = C++, IDE = VS 2010 (64-bit), OS = Windows 7, SP1 (64-bit)
Data Size	*n* = [8192 – 268,435,456], in increasing powers of 2.

Table 8.1: *Histogram Experiment Environment*

All tests were performed on a GTX 580 using the exact same computational environment (shown in Table 8.1). Timings are averages derived by using 100 runs of each test. I/O throughput is measured in gigabytes per second (GB/s) by counting the number of bytes processed in billions and dividing by the average time to process in seconds.

8.4.1 Direct Comparison

In this section, I directly compare my TRISH method to Podlozhnyuk's and Nugteren's methods.

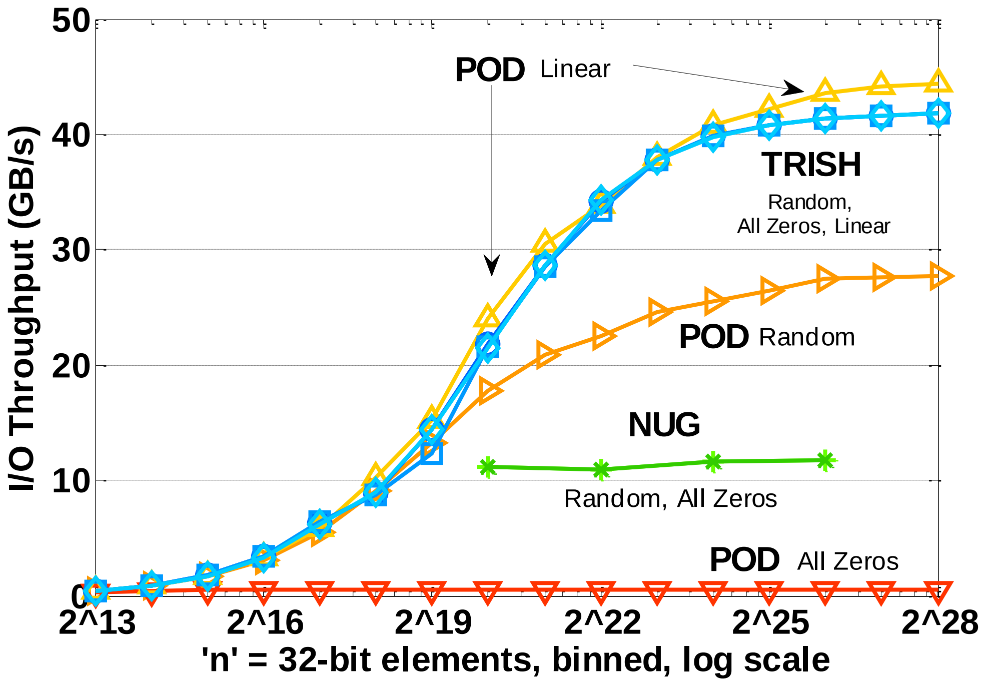


Figure 8.9 *Synthetic Data Throughput*: Throughput for Podlozhnyuk's (POD), Nugteren's (NUG), and my TRISH methods on three synthetic datasets -- *all zeros*, *uniform random*, and *linear*. I base comparisons on uniform random results, although these will be optimistic for POD. Performance measurements were taken on a GTX 580 card, all software was built using the CUDA 4.0 platform; results on other tested platforms were similar.

Because Podlozhnyuk's method is data dependent, Figure 8.9 shows results for three different data sets:

- *All zeros*: A worst-case dataset of "all zeros," which for Podlozhynuk's method causes all threads in each data warp to collide.
- *Linear*: A best-case linear dataset, carefully constructed to prevent any thread collisions in Podlozhynuk's method.
- *Uniform Random*: A *uniform random* dataset, where each data warp should cause a unpredictable number of thread collisions in Podlozhynyuk's method, which according to Raab (Raab and Steger, 1998) should average around 3 collisions per data warp (2.79 = ln(32)/ ln(ln(32)).

The performance results for TRISH and Nugteren's methods on these three different datasets cannot be easily distinguished, confirming their data independence.

Test Conditions: For Podlozhnyuk's method I enabled hardware atomics, which improved performance by 50% over his original software tagging approach. Nugteren's original method is statically compiled for a fixed 2048×2048 image size. I tweaked Nugteren's code slightly to also allow sizes that are 4, 16, or 64 times larger. Generalizing the code further would require range checks that would decrease Nugteren's throughput. The TRISH method bins 31 elements per-thread (k = 31) before looping, for a cooperative stride factor of 95,232 data elements across all 48 thread blocks in the CTA. The grid size was set to 48 to give 3 concurrent blocks per SM on all 16 SMs on the GTX 580. My experiments (not shown) on larger grid sizes resulted in a larger last row size, more range checking, and slightly slower overall performance.

***Data Results*:** The TRISH method has better throughput than the previous methods for large n. For example for $n = 67{,}108{,}864$, TRISH achieves 40.53 GB/s. Nugteren's method achieves 11.25 GB/s. Unlike the other two, Podlozhnyuk's method is highly data dependent: for best-case, uniform random, and worst-case datasets achieves 43.61 GB/s, 27.43 GB/s and 0.48 GB/s, respectively. For uniform random data, TRISH is 3.6× and 1.48× faster than Nugteren's and Podlozhnyuk's methods, respectively.

***Image Experiments*:** In Figure 8.11, I compared all three methods on three standard grey-scale images (Lena, Raft, MRI) obtained from scien.stanford.edu/pages/labsite/scien_test_images_videos.php.

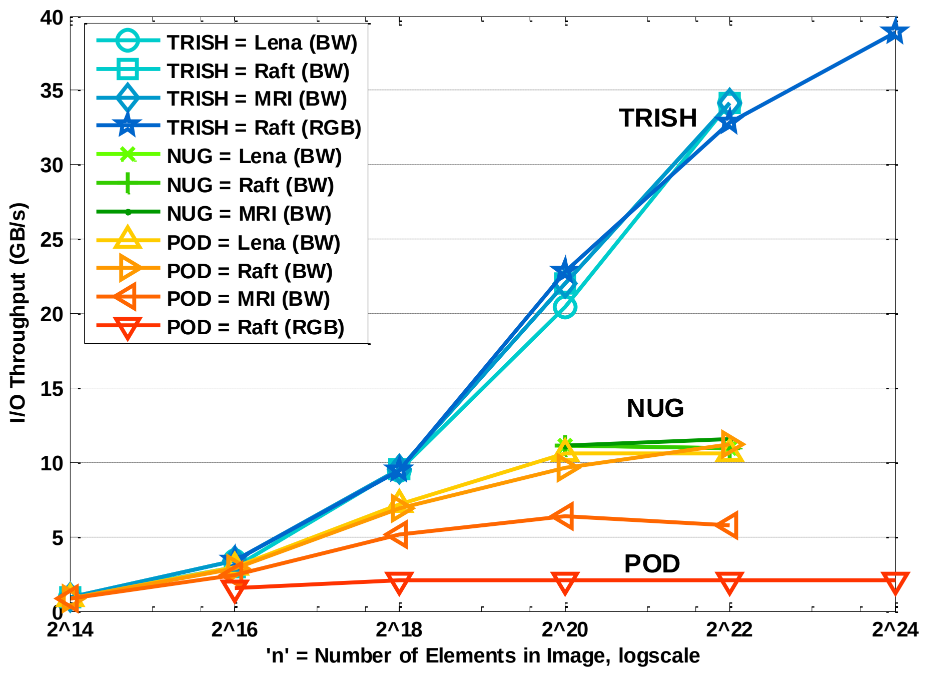


Figure 8.10 *Image Data Throughput*: Throughput for Podlozhnyuk's (POD), Nugteren's (NUG) and my TRISH methods on 3 greyscale images (Lena, Raft, MRI) and one RGBA image (Raft). Each is tiled to sizes 256^2 - 4096^2. Performance measurements were taken on a GTX 580 card, software built on the CUDA 4.0 platform.

As shown in Figure 8.10, Images are tiled to make sizes from 256^2 through 4096^2. I included an RGB Raft image for comparison, which was actually stored as RGBA with the alpha channel set to all zeros (a near worst case for Podlozhnyuk's method.) For all images, the coherence of nearby pixel values hurts the throughput of Podlozhnyuk's method but does not affect the other two. These results indicate that my synthetic uniform random experiment was overly optimistic for Podlozhynuk's practical performance.

I found that my TRISH method was from 1.05× to 4.4× times faster than Podlozhnyuk's method for black and white images, depending on the image and image size. The TRISH method was 2-3× faster than Nugteren's method for the images and image sizes tested. The RGB results underscore the potential danger of using a data dependent histogram method.

8.4.2 Degraded Self-Comparison

I ran three degraded performance tests to determine the impact of ILP and TLP on the overall performance of TRISH.

- *Turn off Row-Sums*: In the first test, I turned off accumulation (row-sums) completely, which makes the results incorrect, but gives an upper bound on performance if one could reduce the frequency of accumulation. Performance increased about ~20% (from 40.9 to 49.4 GB/s, a difference of 8.6).

- *Decrease per-thread Work*: In the second test, I degraded ILP by reducing the amount of work per-thread (k). I measured 18.9 and 40.9 GB/s for $k = 1$ and $k = 31$ respectively. To do this comparison correctly, one must ignore the cost of accumulation. Adding in the accumulation difference of 8.6 from the first degraded test result, I obtained 27.5 and 49.5 GB/s respectively, so TRISH's use of ILP makes the code run roughly two times as fast (49.5/27.5).

- *Decrease Concurrent Thread Blocks*: In the final test, I degraded TLP by reducing the number of concurrent thread blocks per SM from 3 thread blocks per SM to 2 to 1 by padding shared memory. I measured 40.9, 28.5, and 14.6 GB/s for 3, 2, and 1 concurrent blocks per SM, respectively. Consequently, TRISH use of TLP makes the code run roughly three times[8] as fast (40.9/14.6).

8.4.3 Profiler Comparison

I also ran all three methods (POD = Podlozhnyuk, NUG = Nugteren, TRISH = my method) under the NVIDIA Compute Visual Profiler (CUDA 4.0) on a GTX 580 card with *n* = 67,108,864 on uniform random data to gather performance metrics. The results are summarized in Table 8.2.

[8] I expect TRISH could run even faster if there was enough shared memory to support even more concurrent thread blocks per SM – Fermi and Kepler support up to 8 and 16 maximum concurrent thread blocks per SM

	Time (ns)	Through put (GB/s)	Through Put Perf. Ratio	Registers	Shared Mem.	Occupancy	Warps/ Cycles	Ins./ Bytes
TRISH	6,545.41	40.02	**1.00**	36	16,384	192/1536	5.94	11.71
POD	9745.44	27.03	**1.48**	18	6,144	1536/1536	44.47	23.71
NUG	22,631.20	11.25	**3.56**	42	16,384	96/1536	2.98	6.11
	IPC (2.0 Max)	**II**	**Bank Conflicts**	**Cycles II/IPC Ratio**	**Cycles Perf. Ratio**	**Branches**	**Divergent Branches**	**Instruct Replays**
TRISH	1.18	6,140,046	0	**5,203,429**	**1.00**	8,844	0	0.00%
POD	**1.58**	**12,292,604**	2,469,684	7,780,130	**1.50**	1,663,119	940,361	20.09%
NUG	**0.34**	**6,039,095**	7,680	**17,762,045**	**3.41**	1,677	0	8.32%

Table 8.2: CUDA Profiler Results for TRISH, Podlozhnyuk's(POD) and Nugteren's (NUG)

As seen in Table 8.2, all three histogram methods are compute-bound with an *instruction:byte* ratio (Ins/Byte) above the 4.00:1 balanced instructions per byte ratio recommended for a GTX 580 (Micikevicius, 2010, Analysis-Driven Optimization).

This figure shows that Podlozhnyuk's method excels at TLP. With low register and shared memory usage and 192 threads per-block, this method achieves full occupancy of 48 warps ($100\% = 1536/1536$) threads per SM. Such a high occupancy allows the hardware to keep the instruction pipelines busy, retiring 1.58 IPC on average (compared to a hardware maximum of 2.0). However on the negative side, the code in this method issues many more instructions, has many unavoidable bank conflicts, and is involved in a great deal of divergent branching—all of which slow down overall performance dramatically. These performance issues are all directly a result of the intra-warp thread collisions which are resolved using atomics.

Nugteren's straight-forward code issues only half as many total instructions as Podlozhnyuk's. However, Nugteren's method does a poor job keeping the hardware pipelines busy, with only 0.34 IPC, due to small number of bank conflicts, but primarily due to poor TLP caused by low occupancy (6.25% = 96/1536).

Although the TRISH method also has poor TLP due to low occupancy (12.5%=192/1536), it is double that of Nugteren's method. Nonetheless, by taking advantage of

ILP via loop unrolling and batching, TRISH manages to retire 1.18 IPC. This result is not quite as good as Podlozhnyuk's method but is about 3.5× times better than Nugteren's. Despite the extra overhead of having to accumulate regularly, the total instructions issued (II) is competitive with Nugteren's method, due to streamlining code and using some BLP to reduce the number of arithmetic operations. In summary, the TRISH method runs efficiently with no bank conflicts or divergent branching requiring instruction replays.

***Total Cycles*:** So why is the TRISH method faster overall despite being in second place for both for IPC and II? Although it is not immediate apparent, this is because the total cycles required to complete the TRISH method is much lower than either Podlozhnyuk's or Nugteren's methods. Recall that total cycles is computed as TC = II/IPC. As discussed back in Chapter 6 (Case Study on Reduce/Scan), there is an inverse relationship between total cycles and throughput, so a lower total cycles implies a higher throughput result.

The TRISH method beats the Nugteren method by using about twice the TLP and increased ILP to keep the hardware pipeline busy, as captured by the IPC counter. This usage allows TRISH to retire 3.5× more instructions per cycle. The TRISH method beats Podlozhnyuk's method by simply issuing many fewer instructions, as captured by the II counter. This reduced II result is mainly because there are no thread collisions causing instruction replays. Also, I reduced overall computations via bit-level parallelism.

8.5 Conclusion

The 256-bin GPU histogram TRISH method presented here outperforms prior GPU histogram methods, Podlozhnyuk's and Nugteren's, which focused on minimizing I/O via coalesced memory accesses, on storing bin counts in shared memory, and on using simple straight forward binning code. The TRISH method improves compute performance by reducing the total cycles required, by increasing instructions retired per cycle (IPC), by increasing occupancy (TLP) via data compression and register arrays, and by reducing instructions issued (II) via a combination of TLP and BLP. ILP is increased by using *k* work-items per thread. BLP is increased by treating 4 bytes stored as a DWORD as two alternating pairs instead of four singletons.

The final best results of the GPU TRISH method are up to 3.6× times faster than Cedric Nugteren's per-thread method and up to 1.5× times faster than Victor Podlozhnyuk's per warp method for uniform random data. And the results are up to 4× and 3× faster than Podlozhnyuk's and Nugteren's methods, respectively, for image data. Note also like Nugteren’s method, the TRISH method is data independent.

In summary, GPU TRISH is a data independent 256-bin histogram method that achieves a throughput of up to 41.87 billion bytes binned per second (GB/s) on the GTX 580.

8.5.1 Future Directions

My TRISH method falls well short of peak I/O throughput (~21%). Consequently, clever algorithmic improvements, better I/O access patterns, or leveraging of other GPU hardware features can lead to yet more performance gains. Three improvements I have thought about are generalizing TRISH, increasing IPC, and leveraging FPU’s on each SP core.

Generalizing**:** My TRISH method is hardcoded for 256-bin histograms on 8-bit data. It should be possible to generalize it for smaller numbers of bins [1..254] and to support different data types (for instance floats) at an additional arithmetic cost required to distribute values into bins. However, because of shared memory pressure, it will be difficult to generalize the TRISH method to support larger numbers of bins (>256).

Improving IPC**:**..Different versions of the binning and accumulating code could improve ILP even further and achieve a higher IPC—perhaps even equaling or exceeding Podlozhnyuk's IPC of 1.58. If so, overall performance could be increased by up to 33% or more.

Leveraging FPU**:** Each Fermi SM contains 16 used ALUs and 16 unused FPUs representing the SPs on each SM core. It may be possible to use the currently unused FPU hardware for accomplishing additional work during binning and accumulation. For example, binning four additional counts using floats (on FPUs) per 16 counts binned using integers (on ALUs).

8.6 Acknowledgements

I thank the reviewers of my original TRISH paper (Brown and Snoeyink, 2012) for their helpful comments on presentation and suggestions for additional experiments. One reviewer in particular helped me find a copy/paste error that artificially restricted throughput of Podlozhnyuk's method in my initial analysis.

My thanks to NVIDIA, Victor Podlozhnyuk, and Cedric Nugteren for providing their histogram source code online. In that same spirit, I provide a link to my source code below.

TRISH source code: https://github.com/shawndb/demoTRISH

9.0 Case Study: *Radix Sort* on the GPU

In this chapter, I demonstrate my Row *data access skeleton* (DASk) on a *least-significant digit* (LSD) *Radix Sort*. A radix sort produces a sorted output sequence from an unsorted input sequence of *n* keys, which are numbers in some base or radix. My hybrid CPU/GPU LSD radix sort implementation assumes 32-bit keys with 4-bit radices. The CPU implements the main radix sort loop in $k = 8$ (=32/4) passes on the digits which make up each key (one pass per digit). For each pass, the GPU distributes the *n* keys into sixteen sorted runs using a stable *Counting Sort* implemented as three GPU kernels. Achieving solid performance on all three kernels is one of my main goals in this case study.

Knuth (Knuth, 1998) credits Herman Hollerith with inventing radix sort to conduct the 1890 United States census (Hollerith, 1889). Individual census records were represented by punch cards, each with a fixed number of columns (*k*), where each column was represented by a single *digit* in a small range [0, *d*). Hollerith's radix sort fully sorted the final card stack by proceeding in *k* binning passes from the least to the most significant column. In each of *k* binning *passes*, Hollerith's tabulating machine distributed the card stack into *d runs* by directing each card to the bin for the digit value in the currently selected column. At this point a human operator collated the card runs from the bins in monotonically increasing order, reloaded them into the machine, and advanced the machine to the next column to prepare for the next binning pass. After *k* passes, the card stack was fully sorted. A more modern presentation of the serial LSD Radix Sort algorithm is found in on the book "Introduction to Algorithms" (Cormen et al, 2009). In general, LSD radix sort loops over digit positions, from least to most significant, and applies a stable sort comparing that digit in all keys. (Recall that a sort is stable if elements with equal keys are kept in the same relative order.)

There is also a *most-significant-digit* (MSD) radix sort, which bins the card stack on the most significant digit into sub-stacks and then recursively bins each sub-stack separately on the next-most significant digit, and so on. MSD radix sort fragments each stack into many smaller sub-stacks with as many as $\Theta(d^{k-1})$ sub-stacks after k passes on d digits. Since LSD Radix sort setup and storage costs are linear[1] in k, rather than exponential, LSD is more suitable than MSD radix sort for parallel applications aiming for high throughput.

Here follows some useful definitions of common radix sort terms. A *key* is a numeric value taken from a large fixed-size numeric range $[0, m)$. In my GPU radix sort example in this chapter, the n keys are 32-bit unsigned integers in the range $[0, 2^{32})$; so it may be more convenient to work with the number of bits $B = 32 = \lceil \log_2 m \rceil$. A key is written in a base, or *radix*, as a string of *digits*, where each specific digit value is taken from a small fixed-size numeric range $[0, d)$. Typically, the chosen radix is much smaller than the key's range, $d \ll m$. Again, with my chosen radix $d = 16$, it may be more convenient to call it $b = 4$-bits. The maximum number of digits in a key is $k = \lceil \log_d m \rceil$, or, when m and d are both powers of two, $k = \left\lceil \frac{B}{b} \right\rceil$. In my example, there are at most $\leq 32/4 = 8$ digits in each key. The maximum number of digits, k, is also the number of binning passes required in a LSD radix sort. In the jth pass ($0 \leq j < k$), the *chosen digit* is the jth digit within the key used to perform the current pass's Counting Sort to bin n keys into d runs. Within each binning pass, a *run* r_i is a sequence of keys where the chosen digit has the value i, with $i \in [0, d)$. In my example, the r_5 run contains all keys where the chosen digit was equal to 5.

LSD radix sort has often been implemented as a hybrid CPU/GPU algorithm, with a main radix sort loop implemented on the CPU and a stable sorting sub-routine (typically Counting

[1] For a serial LSD radix sort, the total fixed setup cost is $O(k \cdot d)$ to zero out and scan the count histogram on each Counting Sort pass, where k is the number of digits within the key and d is the number of digits in the radix. The total storage cost is $O(n+d)$ to store the input and output arrays of n elements plus an additional array of d counts to hold the count histogram.

Sort) implemented using multiple GPU kernels (Harris et al, 2008; Ha et al, 2009; Merrill and Grimshaw, 2010 Revisiting Sorting).

Counting Sort sorts numbers into runs in three steps, using primitives that generalize those in my previous case studies:

Step 1) **Count Keys** partitions the *n* keys into *d* bins on a chosen digit, by accumulating each digit into its corresponding bin as a count. The *d* bin counters collectively form a count histogram. This counting step is similar to Reduce (see Section 6.1) but generates *d* bin counts instead of a single total sum.

Step 2) **Scan Runs** exclusively scans the histogram of *d* counts into *d* starts. Each *count* (c_i) represents the length of the i^{th} digit run. Each *start* (s_i) represents the starting run position of the i^{th} digit run.

Step 3) **Distribute Keys** distributes the *n* keys into their corresponding digit runs by extracting the chosen digit (with value *i*) from the current key, looking up the current start s_i (run position) at index *i* in the current start histogram, and then incrementing that start (s_i += 1). This distribution step is similar to Scan (see Section 6.1) but works on *d* runs instead of a single run representing a prefix sum.

Steps 1 and 3 take $O(n)$ time, and step 2 takes $O(d)$ time. Consequently, a complete Counting Sort takes $O(n+d)$ time. Because a simple LSD radix sort with *k* iterations (one pass per digit in the key) invokes a Counting Sort on each pass, the total time to compute a LSD radix sort is $O(k(n+d))$. When *k* and *d* are considered constants, the total time to sort is reported as linear, $O(n)$. By binning digits and concatenating runs, radix sort escapes the well-known $\Omega(n \log n)$ lower bound on comparison-based sorts.

My solution implements Counting Sort following the same three-step approach above via three matching GPU kernels -- `GPU_CountKeys`, `GPU_ScanRuns`, and `GPU_DistributeKeys`. To ensure stable sort results, the `GPU_DistributeKeys` must preserve sequential access across data runs, data blocks, and the entire data input array. Since my Row DASk supports sequential access, I use it for both the `GPU_CountKeys` and `GPU_DistributeKeys` kernels. Using the same Row DASk helps ensure that thread blocks in both kernels group and partition data in the same way, which keeps data rows in sync between these two kernels.

In the previous chapters, readers have already learned to improve performance on GPUs by experiments on thread-level parallelism (TLP) via multiple thread warps and on instruction-level parallelism (ILP) via multiple work items. Readers have also learned to respect coalescence, take advantage of multi-issue, avoid register spills, mitigate bank conflicts, and mitigate branch divergence. Therefore, I say little about these good performance practices and drill down on new tips and tricks to improve radix sort performance in a Row DASk, namely reducing I/O, reducing instructions, and increasing occupancy.

9.1 Related Work

The three broad performance themes of reducing I/O, reducing instructions, and increasing occupancy can be found among the details of four papers that influenced my GPU parallel radix sort.

Chatterjee et al.'s Parallel Radix Sort:

In their paper. Chatterjee (Chatterjee et al, 1990) described how to perform a parallel LSD radix sort on a Cray vector computer as a serial/parallel hybrid. They implemented a loop on the serial platform with 32 passes to sort 32-bit integer keys using a 1-bit radix. In each pass, keys were sorted on the current bit using a `split` operation that first performed a parallel scan on the parallel platform using their `+-scan` primitive on the extracted zero bits then performed another `+-scan` on the extracted one bits. Each scan returned output offsets that indicate where to store the scanned keys within each run. Finally, keys were distributed into their sorted locations based on their current radix value {0|1} and their scan offset within the matching run. For fewer I/Os, the authors swap the input and output pointers rather than copying the output back onto input for the next sorting pass.

The authors claim that this parallel radix sort was only 20% slower than a highly optimized comparison-based sort implemented on the same vector architecture. I personally expected an $O(n)$ radix sort vs. an $O(n \log n)$ comparison sort to be faster, not slower. However, by using a 1-bit radix, this radix sort maximizes the number of passes (k = 32) required to fully sort 32-bit keys. In addition, since each split operation does two parallel scans and one parallel permute to distribute keys, the entire radix sort requires a total of 64 parallel scans and 32 parallel permutes. Since each of these scan and permute operations require memory transfers to load inputs and store outputs, this radix sort requires a minimum of at least 192 I/O memory accesses per 32-bit key to fully sort. Better performance might have been obtained by increasing the radix size to reduce the total number of passes, operations, and memory accesses required.

I use a 4-bit radix in my own GPU radix sort implementation primarily to reduce the number of passes[2] (and associated I/Os) from 32 down to 8. I also use Chatterjee's idea of swapping input and output array pointers to reduce CPU I/Os.

Govindaraju's GPU TeraSort:

In 2005 Govindaraju (Govindaraju et al, 2005) won the "Penny Sort Benchmark" contest with *GPU TeraSort*, which sorts 100 byte keys[Sortbenchmark.org] in two steps. The first step uses a hybrid CPU/GPU in-memory sort algorithm to generate sorted runs of data in memory (as much data as would fit). This hybrid sort implements a MSB[3] 4-byte radix sort on the CPU, which in turn invokes a GPU kernel for sorting 4-byte keys. The GPU kernel cleverly tricked the GPU's graphics rendering pipeline into performing a bitonic sort while rendering 4-byte pixel fragments. The second step merges sorted runs using an external, disk-based merge sort algorithm. The authors claimed GPU throughputs of 50 GB/s (or 14 giga-operations per second), and that their overall results were 4× faster than the previous record holders.

[2] I also implemented two GPU radix sorts using 8-bit radices to reduce the number of passes from 32 to 4. However, the performance at 500+ million ‹*key*,*value*› sorted pairs per second was simply not as good as the 700+ million pairs per second for the 4-bit radix results discussed in this chapter.

Harris et al.'s Radix Sort:

In 2008, Mark Harris (Harris et al, 2008) suggested adapting their parallel GPU Scan primitive to implement Chatterjee's parallel radix sort. They used a block-level scan, based on Blelloch's work-efficient scan pattern, to sort individual data blocks, and a higher level hybrid CPU/GPU parallel scan, based on a recursive scan-then-fan pattern (as described in Section 6.4.2), to scan large data arrays on the GPU.

In my experience, this approach to radix sort was difficult to get working correctly and had relatively poor performance on the GTX 280 (15+ million pairs/sec) due to a high number of passes (using a 1-bit radix), the inefficiency of the recursive scan-then-fan pattern used in each pass (as discussed in Section 6.4), and the inefficiency of Blelloch's scan when used on small runs of data.

Ha et al.'s Radix Sort:

In 2009, Linh Ha (Ha et al, 2009) wrote a fast general LSD Radix sort that was also a CPU/GPU hybrid. The CPU performed the LSD radix sort with a 2-bit radix (digit values [0-3]) on 32-bit keys, requiring 16× passes rather than the 32× of Chatterjee's or Harris's methods. A Counting Sort on 2-bit keys is implemented using four GPU kernels as follows:

Kernel 1) An "order checking kernel" performed an "early exit test" on all *n* inputs in each pass. The authors claim this kernel decreased performance by only 2% and allowed the radix sort to exit early if the data was ever found fully sorted. The remaining three kernels followed the two-level scan-then-fan pattern (as described in Section 6.4.2).

Kernel 2) A "4-way radix count kernel" extracted and binned 2-bit keys into per-thread count histograms that were then scanned into per-block starts (using Blelloch's scan) with an extra step to store the four block totals for each scanned data block. The keys were then shuffled (locally sorted) to increase coherence. The entire locally sorted data block was then written back out as well as the per-block sums for the entire data block (4-way starts).

Kernel 3) A "scan kernel" read and scanned all the per-block sums to generate globally correct run positions for each data block.

Kernel 4) A "mapping kernel" loaded the 4-way starts for each data block and then reread the locally sorted data block and then distributed the locally sorted keys into their globally sorted positions.

The authors claimed throughput performance of 60 million key/value pairs sorted per second on the GTX 280. The authors' five main contributions are as follows:

- Increasing the radix to 2 bits, resulting in fewer passes, fewer I/Os, and fewer operations to count and distribute keys, and increasing overall performance.
- Terminating early, if the data became sorted after just a few passes.
- Scanning an entire data block of 4-way counts using a modified version of Blelloch's scan method.
- Generalizing data, the authors used C++ templates to support multiple data types and also discussed specialized techniques to convert floats into unsigned integers while preserving the correct sorting order.
- Shuffling data (local sorting) in shared memory (kernel 2) before mapping the data (global sorting) into its final sorted position in global memory (kernel 4). This increases the coherence in the data stream to increase coalescence on output.

Early versions of my radix sort code also had support for early termination, which I later removed due to performance problems, but my current code contains other ideas from Dr. Ha's paper, including generic programming using C++ templates and shuffling before mapping.

Merrill's Radix Sort:

In 2010, Merrill (Merrill and Grimshaw, 2010, Revisiting Sorting) at the University of Virginia adapted various efficient strategies for radix sort on the GPU to achieve near-peak throughput, up to 480 million key/value sorted pairs per second on the GTX 285. They also used a hybrid method where the LSD radix sort was implemented on the CPU and the three steps of counting sort were implemented on the GPU as three kernels. These three kernels carried out a two-level reduce-then-scan pattern (described in Section 6.4.2.) as follows:

Step 1) **Count**: The “count kernel” creates a histogram for all keys assigned to each thread block. The number of thread blocks within a grid is deliberately kept small to ensure that the next kernel can use a single thread block.

Step 2) **Scan**: The “scan runs kernel” uses the reduce-then-scan pattern (as described in Section 6.4) at the kernel level to convert per-thread block counts into per-thread block start in three steps. The authors call this 3-step method a *block multi-scan* since they are effectively scanning sixteen rows of counts at once for a single data block using a single thread block in parallel. The block multi-scan also needs to scan the resulting sixteen row counts and update all elements in each of the sixteen scanned rows with the correct row prefix for correct per-thread run positions within each data block.

Step 3) **Distribute**: The “distribution kernel” uses a nested reduce-then-scan pattern (as described in Section 6.4) which calls the block multi-scan method on each data block to distribute keys (and values) to their appropriate place in the sorted output arrays.

Merril and Grimshaw’s approach has the following seven main advantages:

• It decreases sorting passes by using a larger radix (4-bit = 16 digit values) than previous GPU Radix Sorts; this decreases the total number of counting sort passes required from 32× to 8×.

• It decreases I/Os by using A 2-level reduce-then-scan pattern (which decreases I/Os from ~4 to 3 per data element on each pass, and results in a fixed number of kernel launches $O(1) = 3$) as compared to a recursive scan-then-fan pattern.

• It scans data block using a block multi-scan method on each fixed-size data block following a nested reduce-then-scan pattern.

• It sorts output locally in shared memory then distributes keys (and values) into global memory for increased coherence and thus increased throughput on output[4].

• It compresses data in arrays in shared memory to increase occupancy.

• It mitigates bank conflicts using the “Pad and Rake” technique (see Section 6.7.2).

• It stores short arrays in registers to increase occupancy.

I use many of Dr. Merrill’s ideas in my own implementation: specifically, his idea of a 2-level reduce-then-scan pattern across my three GPU kernels, a multi-scan on fixed-size data blocks, his efficient parallel `WarpScan` method, compressing data to save space, and the “Pad and Rake” technique to avoid bank conflicts.

[4] Dr. Merrill calls this “exchange then scatter,” but it is equivalent to Dr. Ha’s “shuffle and map” technique.

9.2 Serial CPU Implementation

Because I want to demonstrate the conversion of a CPU sort to an efficient GPU sort using my Row DASk, let me be briefly introduce my own CPU serial implementations of Counting Sort and LSD Radix Sort, which are described in sections 9.2.1 and 9.2.2.

9.2.1 Serial CPU Counting Sort:

As shown in Figure 9.1, my serial CPU Counting Sort method is implemented in six simple steps, with steps 3-5 corresponding to the three kernels of Counting Sort. Steps 1, 2, and 6 allocate the count histogram and run positions array, set the initial counters in the count histogram to all zeros, and free any allocated resources. In the code, I assume my key and digits sizes are both powers of two, so that I can extract my chosen digit on each pass using simple shift and mask operations instead of slower division and modulus operations.

CountingSort sorts a sequence A containing n integer elements, where each element is in range $[0, d)$.

Input: An unsorted sequence $A = [a_1, a_2, \cdots, a_n]$, and d = range of data elements (*nBins*).
Output: A sorted sequence $S = [s_1, s_2, \cdots, s_n]$ (S is a permutation of A), where S satisfies the standard ordering of integers such that $s_1 \leq s_2 \leq \cdots \leq s_{n\text{-}1} \leq s_n$, Note: Counting Sort is *stable*.

Performance: *Total Computations*: $O(n+d)$; *Total I/Os*: $O(n+d)$. Linear $O(n)$ if $d \leq n$.

Counting Sort

```
   CountSort( S, A, n, d, shift, mask )
01: counts = allocate d integers;  // Step 1: Setup resources
02: runs   = allocate d integers;
03: for j in 1..d                  // Step 2: Zero Counts (in count histogram)
04:   counts[j] = 0;
05: end for
06: for i in 1..n                  // Step 3: Count keys (into bins)
07:   key = A[i];
08:   digit = (key >> shift) & mask;   // Extract chosen digit from key
09:   counts[digit]++;                 // Update run count
10: end for
11: sum = 0;                       // Step 4:  Scan runs (start pos's from counts)
12: for j in 1..d
13:   currCount = counts[j];
14:   runs[j] = sum;                 // Save current run's starting position
15:   sum = sum + currCount;
16: end for
17: for i in 1..n                  // Step 5:  Distribute keys (into runs)
18:    key = A[i];
19:    digit = (key >> shift) & mask;   // Extract chosen digit from key
20:    S[runs[digit]] = key;            // Store key in its output run
21:    runs[digit]++;                   // Update run pos for next key in run
22: end for
23: free runs;                     // Step 6:  Cleanup resources
24: free counts;
   end CountSort
```

Figure 9.1 -The serial *Counting Sort* algorithm: The top row has a brief overview of the expected behavior of counting sort with inputs and outputs. The middle row has the corresponding pseudo-code for the serial counting sort algorithm intended to run on a single-core CPU in 6 steps.

The six main steps of my CPU serial Counting Sort are as follows:

Step 1) *Setup* allocates two helper arrays {counts, runs} each of size d. (lines 1-2)

Step 2) *Zero Counts* zeros d counters in the count histogram. (lines 3-5)

Step 3) ***Count Keys*** counts n keys into a count histogram of run lengths. (lines 6-10)

Step 4) ***Scan Runs*** scans the counted run lengths into starting run positions. (lines 11-16)

Step 5) ***Distribute Keys*** distributes n keys into d runs forming the sorted output. (lines 17-22)

Step 6) *Cleanup* frees the two helper arrays {counts, runs}. (lines 23-24)

Steps 1 and 6 each take constant $O(1)$ time, steps 2 and 4 each take $O(d)$ time, and steps 3 and 5 each take $O(n)$ time for a total linear performance cost, with $O(n+d) = 2n+2d+2$. The stability of my Counting Sort method results directly from my implementation of steps 4 and 5 above, where all d run lengths are scanned sequentially and then all n input keys are distributed sequentially. This sequential processing preserves the relative order of keys with the same digit value within the resulting sorted runs.

9.2.2 Serial CPU LSD Radix Sort:

As shown in figure 9.2, my LSD radix sort algorithm loops over k passes of Counting Sort, where k is the number of digits in the maximum key value (computed as $k = \lceil \log_d m \rceil$ or $k = \left\lceil \frac{B}{b} \right\rceil$). The iteration proceeds from the least significant to the most significant digit. All n keys are counted and then redistributed on each digit's pass.

Radix Sort primitive Sorts an unordered integer sequence A containing n integer elements in k passes, where $k = \lceil \log_d m \rceil$ or $k = \left\lceil \frac{B}{b} \right\rceil$, with B = the maximum bits per integer in a fixed-size range $[0, m)$, and b = the maximum bits per radix in a range $[0, d)$. Note: we assume $d \leq m$.

Input: An unsorted integer sequence $A = [a_1, \cdots, a_n]$.

Output: A sorted sequence $S = [s_1, s_2, \cdots, s_n]$ (S is a permutation of A) under the standard ordering of integers such that $s_1 \leq s_2 \leq \cdots \leq s_n$, Note: Radix Sort never uses any direct comparison operators.

Performance: *Total Computations*: $O(k(n+d))$. *Total I/Os*: $O(k(n+d))$. Linear $O(n)$ if $d \leq n$.

```
LSD_RadixSort<B, b>( S, A, n )
// B = #bits in key, b = #bits in radix, A = input array, n = length of input
  d = $2^b$; mask = d-1;
  k = nPasses = ceil(B/b);
  // Allocate resources
  counts[] = allocate d integers;
  starts[] = allocate d integers;
   in[] = allocate n keys;
  out[] = allocate n keys;
  Copy( in, A, n );
  // Setup Ping-Pong pointers
  ping = in;  pong = out;
  if (isEven(nPasses)
     Swap( ping, pong );
  end if
  // Loop over digits in keys (k passes)
  for (pass = 0; pass < k; ++pass)
    shift = b*(pass-1);
    CountSort( pong, ping, n, d, counts, starts, shift, mask );
    // Move to next digit (pass)
    Swap( ping, pong ); // Ping-Pong
  end for (k passes)
  // Cleanup resources (return results)
  Copy( S, out, n );
  deallocate (counts, starts, in, and out)
end LSD_RadixSort
```

Figure 9.2 LSD Radix Sort: The top row has a brief overview of the expected behavior of LSD radix sort with inputs and outputs. The middle row has the corresponding pseudo-code for the serial LSD radix sort algorithm, which iterates over k passes of Counting Sort.

As can be seen in Figure 9.2, the actual code is a main loop over the k passes of Counting Sort with some additional setup and cleanup code. Since each Counting Sort pass takes linear $O(n+d)$ time and there are k passes, the total time is also linear, with $O(n) = O(k(n+d))$. Since there are at most k passes and since the underlying Counting Sort is stable, the entire LSD radix sort is also stable and correctly sorts numeric keys.

Input-output array swapping: Similar to Chatterjee's radix sort, my Counting Sort interface takes both input and output pointers. My LSD radix sort uses two extra pointers (ping, pong) to track the current input and output arrays for each pass. The code swaps the input and output pointers after each Counting Sort pass (see line 15). The partially sorted output from the current pass becomes the new input on the next pass. This simple pointer swap avoids unnecessary array copies, which would otherwise be required to move the output back onto the input. To make sure the output array always ends up with the correct sorted values on the final pass, I swap the ping-pong pointers one extra time during setup if the number of passes is even (see lines 8-11).

Adapting for the GPU: My CPU serial LSD radix sort implementation is easily adapted to work as part of a hybrid CPU/GPU radix sort. There are four required changes:

- The temporary `in`, `out` arrays of n elements are allocated and freed on the GPU instead of on the CPU (lines 5-6 and line 18).
- The CPU copy methods (on lines 7 and 17) are replaced with equivalent `cudaMemCopy` calls to transfer data between the CPU and GPU as needed.
- The single invocation of the CPU Counting Sort (line 14) is replaced by three GPU kernel invocations for the three kernels that make up my GPU Counting Sort.
- The CTA launch parameters {*grid*, *block*} for each GPU kernel are initialized and setup (between lines 11 and 12).

To save space and to avoid redundancy, I do not show my actual CPU host code with these changes.

9.3 Parallel GPU Implementation

As mentioned, a LSD radix sort only works correctly if the Counting Sort sub-method is stable. Sequentially counting and then distributing keys is one way to ensure this stability. Thus, my GPU Counting Sort implementation uses my Row DASk, since it supports sequential access of data rows and data blocks along each row. Within each data block, my code partitions data into short per-thread runs and then enforces sequential access within runs and across runs on input. The keys within each run are binned into per-thread counts, which are then hierarchically scanned into per-thread starts. Prefixes from prior runs, blocks, and rows are all hierarchically accumulated into each final per-thread start to ensure stable sorted results on output.

My Row DASk, discussed in Section 5.4, also supports warp-alignment and automatic load-balances data across the grid rows. In addition, it provides setup and support for a range-check pattern [‹FIRST?›, ‹MIDDLE*›, ‹LAST?›], which pushes expensive range checks out of the middle section and into the first and last data blocks. This approach amortizes the range check costs over the large middle section. The initial setup also helps support full coalescence by aligning the input array to a data warp boundary.

Building upon the serial CPU Counting Sort and LSD radix sort algorithms (as described in section 9.2), my GPU radix sort is implemented as a hybrid CPU/GPU solution in which the CPU-based radix sort iterates over the GPU Counting Sort in *k* passes. Each Counting Sort is implemented as three separate kernels on the GPU, one kernel per major step of the Counting Sort algorithm (count keys; scan runs; and distribute keys). For this particular example, I choose 32-bit unsigned integers as my keys and a 4-bit radix ($d = 16 = 2^4$), which requires $k = 8$ (=32/4) passes to sort. Since the CPU host algorithm is almost exactly the code of Figure 9.2 with the four GPU changes already described, I focus on the details of my GPU Counting Sort.

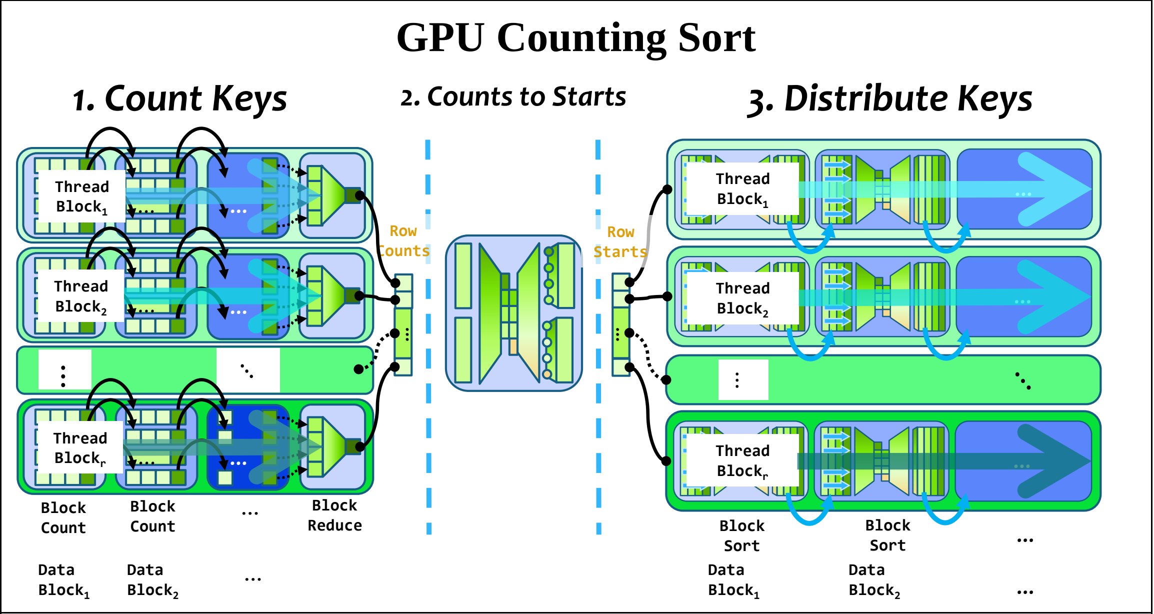


Figure 9.3: My GPU Counting Sort is implemented as 3 kernels (GPU_CountKeys, GPU_ScanRuns, and GPU_DistributeKeys), which correspond to the three main steps of the already described in the CPU Counting Sort (count keys, scan runs, distribute keys). Both the GPU_CountKeys and GPU_DistributeKeys kernels are based on my Row DASk.

In my GPU Counting Sort, depicted in Figure 9.3, the three counting sort GPU kernels (GPU_CountKeys, GPU_ScanRuns, and GPU_DistributeKeys) are based on the two-level reduce-then-scan pattern. (This is similar to how my Scan primitive is implemented using three GPU kernels, see Section 6.5.) The main kernels (the first, GPU_CountsKeys, and last, GPU_DistributeKeys) are implemented using my Row data access skeleton (DASk) from Section 5.4. As before, I choose a small fixed grid size (g < 1000 blocks) so that the middle kernel (GPU_ScanRuns) can convert row-counts into row-starts using a single-thread block.

The GPU_CountKeys kernel (Figure 9.3, left side) generates a total of k (=dp) row counts from n keys using p thread blocks (each thread block generates one count histogram with d row-counts). This kernel uses my Row DASk to group n keys into m fixed-size data blocks (m = $\lceil n/DBS \rceil$) and then partitions the m data blocks across the p thread blocks. This operation initially produces c data blocks per row (c = $\lceil m/p \rceil$). Then, each thread block marches down its assigned data row, block by block, counting keys within each thread block into per-thread count histograms. After all the data within each row has been counted, the per-thread histograms are reduced to a single per-block count histogram (resulting in one row-count histogram per row).

The `GPU_ScanRuns` kernel (Figure 9.3, middle) follows the reduce-then-scan pattern but works on d = 16 rows of data at once. This kernel exclusively scans the k (=dp) row-counts into k row-starts in three main steps using a single thread block. First, the kernel reduces the sixteen rows (of p elements each) down to sixteen final run-counts. Second, the kernel exclusively scans the sixteen run-counts into sixteen run-starts, where each run-start (r_i) is the starting run position of the i^{th} sorted digit run in the output array. Finally, the kernel scans the k (=dp) row-counts into k row-starts. **Note:** This final scan also adds in matching run-starts to the row-starts as prefixes.

The `GPU_DistributeKeys` kernel (see Figure 9.3, right side) distributes keys into their final sorted position in the output array. This kernel uses my Row DASk to group n keys into m fixed-size data blocks ($m = \lceil n/DBS \rceil$) and then partitions the m data blocks across the p thread blocks. This partitioning results in c blocks per row ($c = \lceil m/p \rceil$). Each thread block marches down its assigned data row, data block by data block, distributing keys to their final sorted positions. **Note:** If the programmer is sorting ‹*key*, *value*› pairs instead of just keys, then after distributing a data block of keys, the code also distributes the corresponding data block of values to their final sorted positions.

Both the `GPU_CountKeys` and `GPU_DistributeKeys` kernels must guarantee that each thread block partitions the same data along each data row in order to generate correct results. The easiest way is to use the same Row DASks and CTA launch layouts for both kernels.

In the next subsections, I delve into implementation details for my three GPU kernels.

9.3.1 `GPU_CountKeys` Kernel Implementation Details

The code in Figure 9.4 for my `GPU_CountKeys` kernel (left panel) clearly follows my Row DASk, where data is grouped into blocks and partitioned across rows (one row per thread block). As each thread block moves along its assigned data row, data block by data block, it counts keys into per-thread histograms using the `BlockCount` method (middle panel). After counting all keys in the data row, the kernel accumulates all those per-thread count histograms into a single per-block count histogram using the `BlockReduce` method (right panel).

This is conceptually similar to my `GPU_Reduce` kernel (from Section 6.5), but rather than computing a single total-sum, the `GPU_CountKeys` kernel accumulates multiple counters (one counter per digit value, $d = 16 = 2^4$) at the same time. Because addition is commutative and associative, the code can safely reorder and regroup elements for better I/O throughput.

```
GPU_CountKeys Kernel
Row DASk Setup
// Body Setup
  Zero all counts
for each data block along row
    BlockCount( … )
  Move to next data block
end for each block
// Body Cleanup
  BlockReduce( … )
  Output per-row counts
end GPU_CountKeys
```

```
BlockCount Method
Each thread reads ‹nWork› keys
Each thread bins ‹nWork› keys
  Extract chosen digit from key
  Compute count index
  Compute increment
  Increment counter
end bin keys
end BlockCount
```

```
BlockReduce Method
warpRow = threadID % WarpSize;
fullRows = 16/nWarps;
leftover = 16 – (fullRows*nWarps);
start = warpRow*nDigs;
// Process Full Rows
for each row in [startRow, stopRow]
  Reduce ‹nWarp› run
  Store run-sum in warp array
  WarpReduce run-sums
  Save per-row count (last column)
end for each digit row
// Process Left-Over Rows
if (leftOver)
  if (warpRow < leftOver)
    Same as reducing full row above
  end if
end if
end BlockReduce
```

Figure 9.4: My `GPU_CountKeys` kernel (left panel) follows my *Row* DASk to group *n* keys into *m* fixed-size data blocks and then partitions the *m* blocks across *p* rows (one thread block per row). Each thread block bins keys into per-thread count histograms for all data blocks along its assigned row using my `BlockCount` method (middle panel). After exhausting all keys in the data row, it reduces all the per-thread histograms down to a single per-row count histogram using my `BlockReduce` method (Right panel). The final per-row count histogram is then output to an array of row-counts.

In this kernel, my data layout stores per-thread count histograms (d = 16 counters per thread) in shared memory. (Registers would be faster, but they do not support the necessary indexing.) These histograms are used by the `BlockCount` method to partition work (keys) across all the threads in each thread-block by accumulating intermediate counts. An additional per-block count histogram (d = 16 counters) is kept in shared memory for use by the `BlockReduce` method, which reduces the per-thread count histograms to a single per-block count histogram after counting all keys in the current row. This per-block count histogram is output to the row-counts array.

My `GPU_CountKeys` method requires three types of parameters -- template, function, and CTA launch -- which are described next.

***Template Parameters*:** The following ten template parameters (as shown in Table 9.1) support genericity and experimentation.

Parameter	Explanation
keyT	This parameter is a generic place holder for the underlying data types used to represent keys during the radix sort (U32, for instance).
mapT	This parameter represents a mapping function (or functor) type to convert the actual key types into 32-bit unsigned integers that are consumed by the actual kernel code (for example, floats need to be carefully transformed).
WarpSize	This parameter represents the threads per warp. This value has remained fixed at 32 threads per warp for the past several generations of GPU hardware.
nWarps	This parameter in the range [1-8] specifies how many thread warps per thread block are used. My code only supports values within this range despite the fact that current GPU hardware can actually support up to 32 warps per thread block.
nWork	This parameter in the range [1-16] specifies how many ‹*keys*› or ‹*key*, *value*› pairs are assigned to each thread inside each thread block while counting.
bMap	This Boolean parameter {*true*|*false*} controls whether the *mapOp* transform operator should be applied or not to transform actual keys into 32-bit unsigned integers.
bShift, bMask	These Boolean parameters control whether shifting and masking operations respectively are required to extract 4-bit digits from the 32-bit keys.
Counts PerLane	This parameter was originally intended to experiments using different levels of compression for Bit-level parallelism: However, the value {2} should be hardcoded for now, as it is the only tested and working version.
S1_pad	This flag parameter {0|1} controls whether the "Pad & Rake" technique will be turned on to help mitigate bank conflicts inside the `BlockReduce` method. Since the cost of `BlockReduce` is amortized across all data in the row, this optimization has a negligible performance impact, thus I tend to leave it set to {0} all the time.

Table 9.1: `GPU_CountKeys` C++ template parameters.

Function Parameters**:** The following function parameters (as shown in Table 9.2) are used to count the unsorted keys into a collection of count histograms (one histogram per thread-block):

Parameter	**Explanation**
outRowCounts	This parameter is a pointer to the output array that receives one histogram of row-counts per thread block (data row). Each count histogram contains 16 row-counts (one for each digit value induced by a 4-bit radix, *d* = [0-15]). Each row-count indicates the total count of keys that match the i^{th} digit value along the current data row, where $i \in [0, d)$.
inKeys	This parameter is a pointer to the input array of unsorted keys to be counted as part of the current counting sort pass on the chosen LSD 4-bit digit.
[*start, stop*]	These two parameters identify the range of the input array (*inKeys*) to count. The input size (*n*) can be computed from this range as *n*=*stop*-*start* +1.
keyShift	This parameter indicates how much to shift each key within the transformed 32-bit unsigned key in order to extract the chosen 4-bit digit for the current LSD counting sort pass.

Table 9.2: `GPU_CountKeys` function parameters.

CTA Parameters: The following Grid and Block layout parameters (as shown in Table 9.3) are chosen to support thread-level parallelism (TLP):

Parameter	**Explanation**
Thread Block Size (*BS*)	This parameter is chosen as a 2D block of threads to take advantage of TLP and multi-issue within each SM core. For example: ‹32,4,1› means 4 warps of 32 threads each.
Grid Size (GS)	This parameter is chosen as a 1D grid to load balance the concurrent work load of thread blocks evenly across all the SM cores on the GPU card. For example: ‹1,112,1›). Assume occupancy allows 8 concurrent thread blocks per SM core, then on a GTX Titan, a work load of 112 thread blocks per grid divides 14 SMX cores evenly into 8 concurrent thread blocks each.

Table 9.3: `GPU_CountKeys` CTA parameters.

For the `GPU_CountKeys` kernel, all of the actual counting and reduction work is done by the `BlockCount` and `BlockReduce` methods, which are described in sections 9.3.1.1 and 9.3.1.2.

9.3.1.1 `BlockCount` *Method*:

The `GPU_Count` kernel invokes my `BlockCount` method on each data block to do all of the actual work of counting keys into histograms. The `BlockCount` method (as shown in Figure 9.6, middle panel) has three main steps:

Step 1) **Loading**: Each thread loads ‹*nWork*=[1-8]› keys belonging to the current data block following the warp-by-warp access pattern from Section 5.1.1.

Step 2) **Extracting**: Each thread extracts the chosen digit from each key. Digit extraction is done using simple binary shift and mask operations.

Step 3) **Counting**: For each extracted digit, the thread increments the correct counter in its per-thread count histogram, kept in shared memory.

Since counting (addition) is commutative, sequential counting is not required for correct results. Consequently, the warp-by-warp access block access skeleton (BASk) can be used directly on each data block for maximum coalescence and better throughput (there is no need to convert between a warp-by-warp layout and then a sequential layout within each data block).

Generalizing my `GPU_CountKeys` kernel via template parameters results in various trade-offs. The three main parameters causing trade-offs include: ‹*nWarps*› causing "increased memory use", ‹*nWork*› causing a "counting pipeline bottleneck", and ‹*CountsPerLane*› causing "potential overflow".

‹*nWarps*› causes "Increased Memory Use":

As the number of threads is increased via the ‹*nWarps*› parameter, the amount of shared memory required to store per-thread histograms increases linearly. This memory increase eventually exhausts all memory in the shared memory pool, requiring CUDA to decrease the number of concurrent blocks (and thus warps) per SM core. Once this happens, TLP performance most likely will decrease due to reduced occupancy, negatively impacting throughput.

‹*nWork*› causes "Counting Pipeline Bottleneck":

As the amount of keys per thread being counted is increased via the ‹*nWork*› parameter, the number of RAW stalls caused by counter increments increases linearly. Recall that a simple C++ increment operation requires a read-modify-write resulting in three separate machine instructions. There are RAW dependency between instructions that increment the same counter, which causes pipeline stalls. Trying to remove the stalls via software pipelining causes concurrency bugs, as *k* keys that collide on the same counter should increment that counter by +*k* but instead only increment by +1. Unfortunately, which keys will increment which counters cannot be predicted, so collisions between competing work items (digits) on the same counter within the hardware pipeline are inevitable. ILP performance is decreased due to these RAW stalls. The code shown in Figure 9.5 illustrates this issue.

```
…
// Get indices from digits
LI_1 = (D1*CountSize) + countOff;
LI_2 = (D2*CountSize) + countOff;
…
// Increment counts
// Sequential,
// Correct results but causes RAW stalls
sm_Counts[LI_1]++;
sm_Counts[LI_2]++;
…
```

```
…
// Get indices from digits
LI_1 = (D1*CountSize) + countOff;
LI_2 = (D2*CountSize) + countOff;
…
// Increment counts
// Pipelined, fewer stalls but causes
// Incorrect counts, if LI_1==LI_2*
C1 = sm_Counts[LI_1];
C2 = sm_Counts[LI_2];
…
C1++;
C2++;
…
sm_Counts[LI_1] = C1;
sm_Counts[LI_2] = C2;
…
```

Figure 9.5: Comparing sequential vs. pipelined counting. The left panel shows the sequential method code which is correct but slow due to RAW dependencies hidden within each individual increment. The right panel shows a pipelined method, which is faster but causes incorrect counts due to collisions between keys on the same counter.

‹*CountsPerLane*› causes "Potential Overflow":

I originally intended to enable performance experiments on Bit-level parallelism using the ‹*CountsPerLane* = {1, 2, or 4}›" template parameter. The idea was to support one 32-bit counter per lane, two 16-bit counters per lane (the current default), or four 8-bit counters per lane. This would allow the per-thread count histograms to be compressed saving shared memory (decreasing memory by 1×, 2× or 4× respectively) at the cost of more instructions to compress/decompress counters and the need to prevent overflow when counting. Decreasing shared memory usage enables increased occupancy and thus better TLP performance. Although I have written code for all three approaches, I have only tested the middle case (two 16-bit counters per lane). So, that is what should be used for now.

The actual count histograms are stored as eight lanes of 32-bit values with two 16-bit counters per lane. A 16-bit counter can overflow if it is incremented past 65,535 (=2^{16}-1). So, the code needs to detect and guard against this overflow. My current, potentially unsafe, solution assumes that 16-bit overflow cannot actually occur with my current CTA layout and data set sizes (this assumption limits[5] my input sizes to $n < 2^{27}$). A safer and more general solution, shown in figure 9.6, would be to accumulate per-thread histograms into a per-block histogram on a regular basis using the already existing `BlockReduce` method.

```
// Handle overflow …
currWork = 0;
maxWorkBeforeOverflow = (2^16-nWork);  // Note: use 2^8 instead for 8-bit overflow
while (more data blocks)
  // Overflow possible? (conservative test)
  if (currWork >= maxWorkBeforeOverflow)
    BlockReduce( … );  // Accumulate thread histograms into single block histogram
    Zero 'per-thread' histograms
  end if
  ...
  BlockCount( … );
  ...
  currWork += nWork;
  Move to next data block
end while (data)
```

Figure 9.6: Handle potential Overflow by reducing per thread histograms down to a per-block histogram just before 16-bit overflow (or 8-bit overflow) can happen.

[5] I assume the original maximum input size (n) is 32-bits (0xFFFFFFFFu). I assume both keys and values are 32-bit integers and require both input and output arrays. Since global memory is stored on byte boundaries, this reduces n from 32 to 28 (=32-log(4 arrays *4 bytes)) bits. Furthermore, I assume that data is partitioned evenly across all threads. Assuming at least one thread warp per thread block (32 threads) and a minimum grid size of 128 thread blocks, results in a minimum of at least 4096 threads (=32*128), which reduces the maximum elements assigned to each individual thread from 28 to 16 bits (=28-log(4096)), which almost fits in a 16-bit counter (in the worst case, 65,536 = 2^{16} counts is one value too large for 2^{16}-1), decreasing the maximum input size to n=2^{27}-1 bits avoids overflow under these assumptions.

As may be apparent, this code, after calling `BlockReduce,` would also need to reset all the per-count histograms to zero. The word “regular” means that the code conservatively tests for potential overflow of a loop counter against a maximum overflow variable, where even a single additional iteration would cause the 16-bit counter to potentially overflow. This conservative approach assumes the worst case, where all increments always end up in the exact same counter[6].

9.3.1.2 `BlockReduce` *Method*:

The `GPU_CountKeys` kernel uses my `BlockReduce` method to reduce multiple per-thread count histograms down to a single per-block (row-count) histogram. The goal is to reduce 16 rows of counters (with TBS entries in each row) down to 16 final row-counts in parallel. My code load-balances the row-counts across ‹*nWarp*› thread warps within each thread block, so that each thread warp processes *fullRows*, where *fullRows* = $\left\lceil \frac{16}{nWarps} \right\rceil$. If the number of warps does not divide evenly into the number of histogram entries (d = 16), then extra code is also required to reduce the partial leftover rows (*leftOver* = 16-(*fullRows***nWarps*)). For each full work row of counters, each thread warp loads and reduces ‹*nWarp*› counts per row down to a single count. The single count is then stored in an intermediate array, and subsequently the `WarpReduce` method (see Section 6.6.3) is invoked to reduce 32 single counts using all 32 threads in the warp down to a final row-count. **Note:** The final row-count is found in the last column of each warp reduced array. Next, each final row-count is extracted and stored in another per-block row-count histogram in shared memory. Finally, the resulting 16 final row-counts are output to an intermediate array in global memory, which corresponds to this data row (thread block). The `BlockReduce` method is called rarely to prevent 16-bit overflow, currently only once per data row. So, the performance costs are amortized across all the data blocks along each data row.

[6] For example, an array of input keys that were all set to zero could cause this behavior.

Memory Reuse: There is really only one general issue with my BlockReduce method: reducing shared memory usage to mitigate constraints on occupancy. My solution is to reuse memory: My code allows two different arrays to share the same memory buffer (the per-thread count histograms from BlockCount and the warp-reduce arrays from BlockReduce). These two methods do not overlap in time, since BlockReduce is not invoked until after all data blocks have been counted using BlockCount. This allows the code to safely overlap these two different arrays within the same memory space, which is the larger of these two different arrays (*spaceReq* = max(sizeArray(*thread-counts*), sizeArray(*warp-reduce*)).

9.3.2 GPU_ScanRuns Kernel

My GPU_ScanRuns kernel (as shown in Figure 9.7) is used to scan row-counts into row-startsthat will later be used as starting prefixes for storing sorted runs of keys (and values) by my GPU_DistributeKeys kernel. The GPU_ScanRuns kernel is conceptually similar to my GPU_SumsToStarts kernel, used in my Scan primitive (see Section 6.5). The pseudo-code for my GPU_ScanRuns kernel follows the 2-level reduce-then-scan pattern (described in Section 6.4) but scans 16 different rows of row-counts instead of just one row.

To support the 2-level scan pattern used across all 3 kernels making up my GPU Counting Sort, the GPU_ScanRuns kernel is launched using a single-thread block of TBS threads(gridsize = 1). This means the grid size for both the GPU_CountKeys and GPU_DistributeKeys kernels must be kept reasonably small ($p \leq 1000$). My GPU_ScanRuns has three main steps: reducing to final counts, scanning final counts, and scanning row counts:

GPU_ScanRuns *Kernel*

```
 1:  Row DASk Setup
 2:  Zero local arrays (including TBS counters)
     // Step 1) Reduce (p*d) row-counts down to (d) final counts
 3:  for all row-counts  // in parallel across TBS threads
 4:    Read TBS row counts
 5:    Accumulate TBS row counts into TBS counters
 6:    Move to next data block (+TBS)
 7:  end for
 8:  Reduce TBS individual counters down to (d) final counts
     // Step 2) Exclusively Scan final counts into final starts  // in parallel across d threads
 9:  Scan (d) final counts
10:  Reach back one column to get exclusive results
11:  Save d scanned counts as d start prefixes
     // Step 3) Exclusively Scan row-counts into row-starts
12:  for all row-counts  // in parallel across TBS threads
13:    Read DBS row counts
14:    d-way multi-scan on DBS row counts
15:    Save d block-sums
16:    Write DBS scanned row starts (+ count prefix)
17:    Update count prefixes (count prefixes +=block-sums for i in 0..d-1)
18:    Move to next data block (+DBS)
19:  end for
```

Figure 9.7: `GPU_ScanRuns` Kernel pseudo-code. The `GPU_ScanRuns` kernel follows the reduce-then-scan pattern but performs a *d*-way multi-scan on the *d* entries in *p* count histograms to generate starting run positions for each of *p* thread blocks. This partitions the keys (and values) output locations safely across *p* concurrent thread blocks.

Step 1) **Reduce to final counts:** This first step (Figure 9.7, lines 3-8) reduces all k ($=dp$) row-counts to $d = 16$ final counts. The thread block size (*TBS*) is chosen to be a multiple of the number of digit values (d). For instance, 256 threads is evenly divisible by $d = 16 = 2^4$ for a 4-bit radix. Before accumulating any row-counts, the code first calculates the number of full-rows ($nRows = k/TBS$) and the presence of a last partial row ($leftOver = k - (nRows \cdot TBS)$) assigned to each thread. The full-rows do not require any range checking. The last partially full row requires careful range checks. Each thread then loads and accumulates all assigned row-counts from the full-rows and the last partially full row (if it exists). After accumulating all assigned row counts into intermediate thread-counts, the *TBS* thread-counts need to be reduced to $d = 16$ final counts. This is done by storing all *TBS* thread-counts in a shared memory array and then using $d = 16$ threads to serially accumulate (TBS/d) partial sums each, with a stride of $d = 16$ elements between each row. This step produces $d = 16$ final run-counts.

Step 2) **Scan final counts:** This middle step (Figure 9.7, lines 9-11) is trivial, where $d = 16$ threads inclusively scan $d = 16$ run-counts into $d = 16$ run-starts by invoking the parallel `WarpScan` method. Recall that WarpScan also requires a half-array (8 columns) of identity elements (zeros). However, exclusive (not inclusive) scan results are needed for correct run-starts. Consequently, the code reads exclusive final-starts from the inclusively scanned results by reaching back one column in the warp array. These $d = 16$ final run-starts indicate the starting location of each sorted digit run in the sorted output array and are used as run prefixes to generate correct per-block row-starts in the next step.

Step 3) **Scan row counts:** This last step (Figure 9.7, lines 12-19) scans k (=*dp*) row-starts into k row-starts. A Row BASk access pattern is then applied. Given a fixed data block size (*DBS* = *TBS***nWork*), the code groups the k row-starts into m (=$\lceil k/DBS \rceil$ fixed-size data blocks, and the single thread block marches along this single data row, data block by data block, scanning each data block in turn from DBS row-starts into DBS row-starts. The first m-1 data blocks require no range checks, while the last partially full data block may require careful range checks. For each data block, the code performs a d-way inclusive multi-scan on the *DBS* row-counts, adds in the correct run-start prefix from step 2, and then outputs the d-way exclusive scan results as *DBS* row-starts. After each data block is scanned, the resulting d = 16 block totals are accumulated into the current d = 16 run prefixes from step 2 to prepare for the next data block along the row.

After scanning, the "row-starts" array contains valid block-starts within each sorted run for each individual thread-block. These block-starts safely partition the sorted output array across all thread blocks (with one set of d=16 block-starts for each thread block). The row-starts enable the `GPU_DistributeKeys` kernel to independently distribute its keys (and values) into sorted runs without conflicting with other thread blocks concurrently distributing data at the same time. The `GPU_ScanRuns` kernel has one performance issue in step 3, which is 2-way bank conflicts as described next.

Multi-scan 2-way Bank Conflicts: In Step 3 above, each thread warp is responsible for scanning (2 = *d*/*nWarps* = 16/8) row-counts within each data block. Since 32 threads in a warp can store two sets of d values (32/16), a modified `WarpScan` method (Section 6.6.3) is used, which actually performs two 16-element scans at once. Unfortunately, adding an 8-element pad section before each 16-element scan section immediately results in 2-way bank conflicts between all threads in the same warp. To avoid bank conflicts, I tried re-orienting the data layout and using an extra level of indirection to access the data during the scan[7]. However, this solution was

[7] I tried 2 different solutions, both with poor results: 1) I moved all zeros into the first data warp and then had all scans referenced the sub-set of zeros that resulted in no bank conflicts, an extra level of indexing was required to make this approach work correctly. Unfortunately, this extra level of indirection hurt overall performance. 2) I tried re-using my 3-step conversion method to move data into a new data layout where bank conflicts could not happen. Unfortunately, this required 3× as much memory and 3× as many shared memory accesses, which hurt overall performance.

more computationally expensive than the bank conflicts themselves. Consequently, my current code just lives with the 2-way bank conflicts.

9.3.3 GPU_DistributeKeys Kernel

My GPU_DistributeKeys kernel (as shown in Figure 9.8) distributes keys (and values) into sorted runs based on the current chosen digit. The GPU_DistributeKeys kernel is based on the GPU_ScanRuns kernel used in my Scan primitive (in Section 6.5). The main difference is GPU_DistributeKeys must do a lot more work than just scanning (as in GPU_ScanRuns) to correctly support the distribute phase of my GPU Counting Sort. As already discussed, the code for this kernel uses my Row DASk for sequential processing of data to support stable sorts. As per the Row DASk access pattern, this kernel groups *n* input keys into *m* fixed size data blocks (*m* = $\left\lceil \frac{n}{DBS} \right\rceil$). The kernel then load-balances the *m* data blocks evenly across the *p* thread blocks in the CTA grid (*nBlocks* = $\left\lceil \frac{m}{p} \right\rceil$). Each thread block finally marches down its assigned data row, data block by data block, calling the BlockDistribute method on each fixed-size data block of keys (and values) in sequence.

My BlockDistribute method does a local counting sort in parallel on each data block to determine local offsets (within the current data block) and then adds in matching row-starts to get global offsets before distributing the locally sorted data into their final sorted positions After distributing one block of keys (and values) into *d* = 16 sorted runs, the *d* = 16 row-starts are incremented by the matching *d* = 16 block-sums from the current data block to prepare to distribute the next data block along the row.

`GPU_DistributeKeys` *Kernel*

```
GPU_DistributeKeys
// Distribute all ‹key, value› pairs into sorted runs.
Row DASk Setup
    Zero local arrays  // Distribute Setup
    Load initial run-starts[0-15]
if (rc.Both)
    BlockDistribute_RC_BOTH( … ) ②
end if
if (rc.Start)
    BlockDistribute_RC_START( … ) ②
end if
for each data block along row
    BlockDistribute_RC_NONE( … ) ②
end for
if (rc.Stop)
    BlockDistribute_RC_STOP( … ) ②
end if
```

```
BlockDistribute_RC_*  ②
// Distribute one data block of ‹k,v› pairs into sorted runs.
  Pre-computed constants
   // Bin Keys
  STEP 1.0:  LoadKeys (with range checks)
  STEP 2.0:  ExtractDigits
  STEP 3.0:  CountKeys
   // Scan counts (into starts)
  STEP 4.0:  ScanWarps
  STEP 5.0:  ScanBlocks
  STEP 6.0:  UpdateKeyStarts
   // Sort Keys
  STEP 7.0:  ShuffleKeys
  STEP 8.0:  MapKeys (with range checks)
   // Sort Values (Optional)
  STEP 9.0:  LoadValues (with range checks)
  STEP 10.0:  ShuffleValues
  STEP 11.0:  MapValues (with range checks)
   // Move to next data block (along row)
  STEP 12.0: UpdateRunStarts
end BlockDistribute
```

STEP 1.0: `LoadKeys_RC_NONE`
// Load short run of *nWork* keys [1-8] into registers.
Setups "load", "store", and "run" pointers.
1.1: *Loads* nWork *keys* [1-8] *from* loadPtr.
1.2: *Stores* nWork *keys* [1-8] *into* storePtr.
1.3: *Loads* nWork *keys* [1-8] *from* runPtr.

`LoadKeys_RC_BOTH`
Also range checks array accesses against [start, stop].
`LoadKeys_RC_START`
Also range checks array accesses against [start, …).
`LoadKeys_RC_STOP`
Also range checks array accesses against (…, stop].

STEP 2.0: `ExtractDigits`
// Extract digits [0-15] from keys [1-8].
2.1: (*Optional*) *Map keys into 32-bit unsigned integers*
2.2: *Extract digits from keys* (*shift & mask*)
2.3: *Compress digits into single 32-bit register*

STEP 3.0: `CountKeys`
// Count Keys (into 'per thread' histograms.)
3.1: *Zero counts array*
3.2: *Compute lane indices*
3.3: *Compute shifts and increments*
3.4: *Accumulate counts (save initial starts)*
3.5: *Compress initial starts into 32-bit register*

STEP 4.0: `ScanWarps`
// Scan 'per warp' counts into 'per warp' starts
Setups "S1"scan and "S3" store pointers.
4.1: S1:SS‹4›, *each thread in warp serially scans a short run of* 4 *lane counters.* (4 *8-bit counters per 32-bit lane*).
4.2: S2:SS‹8›, *each active thread* (4 *per warp*) *serially scans a short run of* 8 S1 *run sums. Also stores final S2 run sums into S3 scan array.*
4.3: S1:SU‹4›, *each thread updates short run of* 4 *scanned results with S2 prefix* (*from Step* 4.2)

STEP 5.0: `ScanBlocks`
// Scans 'block' counts into 'block' starts
Setups "S3"scan and "S4" pointers.
5.1: S3:SS‹8›, *each active thread* (8) *serially scans a run of* 8 *lane counters.* (2 *16-bit counters per 32-bit lane*).
5.2: S4:WS‹8›, *all active threads* (8 *total*) *warp scan 16 block sums into 16 block starts.*
5.3: S3:SU‹8›, *each active thread* (8) *updates run of* 8 *scanned results with S4 prefix* (*from Step* 5.2)

STEP 6.0: `UpdateKeyStarts`
// Grab final 'per key' starts
Setups "S1"scan and "S3" pointers.
5.1: *Grabs* [1-8] *per key starts from compressed reg.*
5.2: *Adds in correct S1 prefixes* (*from Step 4.3*).
5.3: *Adds in correct S3 prefixes* (*from Step 5.3*).

STEP 7.0: `ShuffleKeys`
// Sort Keys into shared memory array (Local Sort)

STEP 8.0: `MapKeys` (range checks like STEP 1.0)
// Sort Keys into outKeys array (Global Sort)

STEPS 9.0, 10, & 11: Sort Values
Similar to Steps **1.0 (Load)**, **7.0 (Shuffle)**, *and* **8.0 (Map)** *respectively but with "values" instead of "keys".*

STEP 12.0: `Update Run Starts`
// Add current block sums [1-16] into run starts [1-16].

Figure 9.8 `GPU_DistributeKeys` Kernel: This kernel follows my *Row* DASk. *n* keys are grouped into *m* data blocks that are partitioned across *p* thread blocks. Each thread block marches down its data row, calling `BlockDistribute` on each assigned data block in turn. `BlockDistribute` loads and bins keys into counts (Steps 1-3), scans counts into starts (Steps 4-6), sorts keys (Steps 7-8), optionally sorts values (Steps 9-11), and then gets ready for the next data block (Step 12).

Since my `GPU_DistributeKeys` kernel follows the Row DASk, there is a setup stage at the beginning of the kernel, which initializes the desired Row DASk behavior and also initializes the local arrays used by this kernel. This setup step has four main sub-steps:

Sub-Step 1) **Load Balance:** the kernel computes the *nBlocks* (*m*) assigned to this data row using biased load-balancing, as described in section 5.6.3.

Sub-Step 2) **Range Check:** the kernel initializes the variables used with the ‹FIRST?, MIDDLE*, LAST?› range check pattern, as described in section 5.6.2.

Sub-Step 3) **Zero Arrays:** the kernel allocates and zeros intermediate arrays in shared memory, which are used by the `BlockDistribute` method.

Sub-Step 4) **Load row-starts:** the kernel loads the *d* = 16 row-starts that indicate where each thread block can safely output its sorted data within each output run without conflicting with other concurrent thread blocks outputting data at the same time. These row-starts (starting block run positions) were computed by the `GPU_ScanRuns` kernel.

My `GPU_DistributeKeys` method requires 3 types of parameters--template, function, and CTA launch--which are described next.

Template Parameters**:** The following twelve template parameters (as shown in Table 9.4) are used to support genericity and experimentation for this kernel.

Parameter	**Explanation**
keyT, valT	These parameters are generic placeholders for the different key and value types used to represent the ‹*key*, *value*› pairs (‹U32, U32› for instance).
mapT	This parameter represents a mapping function (or functor) type to convert the actual keys into 32-bit unsigned integers that are consumed by the actual kernel code (for example, floats need to be transformed).
WarpSize	This parameter represents the threads per warp. This value has remained fixed at 32 threads per warp for the past several generations of GPU hardware.
nWarps	This parameter in the range [1-8] specifies how many thread warps per thread block are used. My code only supports values in this range, despite the fact that current GPU hardware can support up to 32 warps per thread block.
nWork	This parameter in the range [1-8] specifies how many ‹*keys*› or ‹*key*, *value*› pairs are assigned to each thread inside each thread block while sorting.
bMap	This Boolean parameter {*true*|*false*} controls whether or not the *mapOp* transform operator should be applied to transform actual keys into 32-bit unsigned integers.
bShift, bMask	These Boolean parameters control whether shifting and masking operations respectively are required to extract 4-bit digits from the 32-bit keys.
bHasValues	This Boolean parameter controls whether to sort ‹*key*, *value*› pairs {*true*} or ‹*key*› singletons only {*false*}.
IO_pad, S1_pad	These parameters control whether the "Pad & Rake" technique will be turned on to help mitigate bank conflicts {1} or left turned off {0}. The *IO_pad* should only be set to {1} when the *nWork* parameter is a power of 2 other than one [2,4,8], otherwise it should be set to zero. Invalid use can lead to kernel crashes or incorrect results.

Table 9.4: `GPU_DistributeKeys` C++ template parameters.

Function Parameters**:** The following eight function parameters (as shown in Table 9.5) are used to distribute the unsorted input arrays into output arrays containing sixteen sorted runs (one run per digit value):

Parameter	**Explanation**
outKeys, outVals	These parameters represent the output arrays that receive the sorted keys and values as part of the current counting sort pass on a chosen LSD digit.
inKeys, inVals	These parameters represent the unsorted input arrays (keys and values) that need to be sorted on the chosen LSD digit as part of the current counting sort pass.
inRowStarts	The *inRowStarts* parameter is a pointer to the scanned row-starts from the `GPU_ScanRuns` kernel. There are sixteen entries per thread block (one for each digit value, d = [0-15]). Each *row-start* indicates the unique starting position for the current thread block to output its sorted output data within the d^{th} sorted run without conflicting with other concurrent thread blocks outputing data at the same time.
[*start, stop*]	These parameters identify the range of the input arrays (*inKeys*, *inVals*) to sort by distributing into runs. The input size (n) can be computed from this range as n=*stop*-*start* +1. **Note:** The output arrays are also assumed to use the same range.
nWork	This parameter in the range [1-8] specifies how many ‹*keys*› or ‹*key*, *value*› pairs are assigned to each thread inside each thread block while sorting.
bMap	This Boolean parameter {*true*|*false*} controls whether the *mapOp* transform operator should be applied or not to transform actual keys into 32-bit unsigned integers.
bShift, bMask	These Boolean parameters control whether shifting and masking operations respectively are required to extract 4-bit digits from the 32-bit keys.
bHasValues	This Boolean parameter controls whether to sort ‹*key*, *value*› pairs {*true*} or ‹*key*› singletons only {*false*}.
keyShift	This parameter indicates how much to shift each key within the transformed 32-bit unsigned key in order to extract the chosen 4-bit digit for the current LSD counting sort pass.

Table 9.5: `GPU_DistributeKeys` function parameters.

CTA Parameters: The following Grid and Block layout parameters (as shown in Table 9.6) are chosen to support thread-level parallelism (TLP):

Parameter	Explanation
Thread Block Size (*BS*)	This parameter is chosen as a 2D block of threads to take advantage of TLP and multi-issue within each SM core. For example: ‹32,4,1› means 4 thread warps of 32 threads each.
Grid Size (GS)	This parameter is chosen as a 1D grid to load balance the concurrent work load of thread blocks evenly across all the SM cores on the GPU card. For example: ‹1,112,1›). Assume occupancy allows 8 concurrent thread blocks per SM core, then on a GTX Titan, a work load of 112 thread blocks divides 14 SMX cores evenly into 8 concurrent thread blocks each.

Table 9.6: `GPU_DistributeKeys` CTA parameters.

For the `GPU_DistributeKeys` method, all the actual work of distributing keys (and values) into sorted runs is done by the `BlockDistribute` method (a method overview and the 12 individual steps of this complex method are described in section 9.4.3.3). Also, there are several general performance issues associated with this method, which are described first in section 9.4.3.1. Other issues unique to each step, are described within each individual step. Some general solutions to these general performance issues are described in section 9.4.3.2. Other solutions unique to each step, are described within each step.

9.3.3.1 `BlockDistribute` *Issues*:

There are three general issues with `BlockDistribute` – bank conflicts, register pressure, and shared memory pressure.

Bank Conflicts: Since each thread works with short sequential runs [1-8] in shared memory, bank conflicts during shared memory accesses become a performance issue. As discussed in section 6.7.2, odd-length runs [1, 3, 5, 7] will not cause any bank conflicts, but even length runs [2, 4, 6, 8] can result in [2-way, 4-way, 2-way, and 8-way] bank conflicts costing [1, 3, 1, or 7] extra machine cycles per shared memory access, respectively. For run lengths that are powers of two [2, 4, 8], the “Pad and Rake” technique can be used to eliminate bank conflicts at the cost of an extra pad column in memory and extra rake instructions to skip over the pad column. Users can, of course, choose to live with the performance costs due to extra cycles caused by the bank conflicts (for example, for runs of length [6]). If you choose to live with the bank conflicts, CUDA's low level SASS compiler may decide to optimize an aligned sequence of two or four 32-bit load (or store) instructions as a single Vector2 (64-bit) or Vector4 (128-bit) instruction. This optimization decreases the total number of instructions and bank conflicts by 2× or 4×, respectively, since instructions that never get executed do not cause bank conflicts

Register Pressure: My code takes advantage of each GPU core's support for instruction pipelining to improve ILP performance by having each thread work on multiple (*k*=[1-8]) work-items at once and then grouping and reordering instructions using software pipelining. Linearly increasing the amount of work per thread by *k*× increases the number of registers per thread needed *O*(*k*) to keep track of multiple keys, counts, starts, etc. throughout the code. Consequently, more work means more registers.

My code also takes advantage of the GPU's support for massive multi-threading to improve TLP performance by launching multiple thread blocks with multiple thread warps (*w*=[1-8]) within each thread block, with each warp containing 32 threads. However, each individual thread needs its own unique set of independent registers to ensure correct behavior. More threads means more registers.

Recall that each SM core contains only a fixed pool of registers (32K or 64K), that are shared across all active thread warps (up to 48 or 64 thread warps per SM). Too much work or too many warps can constrain occupancy due to register pressure, which decreases TLP performance. My BlockDistribute method keeps both the supported amount of work ‹*nWork*› and the supported number of thread warps ‹*nWarps*› small (≤ 8) for my experiments.

Shared Memory Pressure: The `BlockDistribute` method needs several different memory arrays stored, which are kept in shared memory. For instance, shared memory is needed to convert between data warps in global memory (stride = 32) and short sequential runs of keys (and values) kept in registers (stride = 1). Several different memory arrays are also needed to count keys into per-thread count histograms and later to scan those counts into starts (local sorted offsets). Finally, shared memory arrays are needed to sort keys (and values) locally before distributing them globally into sorted runs. Larger data block sizes (DBS) results in larger arrays. Large arrays reduce occupancy due to shared memory pressure, which hurts TLP performance. My solution reuses the same memory buffer for multiple arrays, which is discussed more fully in section 9.4.3.2, Step 12 (Move to next Block).

9.3.3.2 `BlockDistribute` *Solutions:*

To handle bank conflicts, I apply different techniques from section 6.7.2 to different steps in this `BlockDistribute` method. To decrease register pressure, I batch my software pipelining into small groups. To decrease shared memory pressure, I reuse memory to save space.

Handling Bank Conflicts: As already discussed in Section 6.7.2, there are multiple ways to deal with bank conflicts 1) Live with the performance hit; 2) Use runs whose length is odd; 3) Use the Pad and Rake technique on runs whose length is a power of two; 4) Enable the SASS compiler Vector2/Vector 4 optimization by aligning data to a 16-byte boundary. Almost all of the twelve steps in this method result in bank conflicts.[8] Sometimes I live with the performance hit; sometimes I use the “Pad and Rake” technique. I try to align my data arrays in shared memory to 16 byte boundaries to enable the Vector2/Vector4 optimization by the SASS compiler, adding extra pad columns where necessary. Note: The “Pad and Rake” technique prevents the Vector4 optimization by the SASS compiler.

[8] For many of these individual steps, I have tried multiple solutions to mitigate bank conflicts. For instance, early versions of my radix sort code used my 3-step conversion technique from `LoadKeys` over and over again between steps to eliminate as many bank conflicts as possible. However, one of my primary goals is to increase throughput by decreasing total cycles. So, I only kept solutions which increase data throughput. As a result, the 3-step conversion technique got dropped as it was usually more expensive then living with the bank conflicts it removed.

One interesting wrinkle is that avoiding bank conflicts in one step may cause them in another step. For instance, step 3 (`CountKeys`), step 4 (`ScanThreads`), and step 6 (`AccumulateLocalStarts`) access the same memory buffer using two different views. In this case, I try it both ways and see which way has the best performance.

Another wrinkle is that some bank conflicts are simply unavoidable. For instance, Step 7 (`ShuffleKeys`) results in an unpredictable number of bank conflicts, since the algorithm cannot predict ahead of time which bank column in shared memory each key will be stored into. Based on a 32 balls into 32 bins simulation on 10,000 runs on uniform random data, the average number of bank conflicts is about 3.4 on each memory access per thread warp.

Compression: To save registers, instructions, and shared memory, I compress data. For instance, I compress up to eight digits into a single 32-bit register, which reduces register storage costs by a factor of 8×. Also, when counting, I compress four 8-bit counters within a single 32-bit lane, which reduces the number of registers and shared memory required to store per-thread count histograms by a factor of 4×. However, there is a trade-off here, since many more instructions and temporary registers are also required to correctly compress and later decompress these values as needed. Where possible, bit-level parallelism is taken advantage of to decrease the number of total instructions required. For example when scanning, the code can scan four (or two) compressed runs at the same time using one set of serial Sklansky scan instructions, reducing the number of scan instructions required by a factor of 4× (or 2×).

Batched Software Pipelining decreases Register Pressure: Software pipelining *k* work items increases the number of required registers by *k*×. Range checked I/O steps (such as steps 1.0, 8.0, 9.0, and 11.0) require extra registers for the checks. Other steps (such as 3.0, 4.0, 5.0, and 6.0) require complex logic to implement correctly, which also tends to increase the number of registers required. For these complex steps, one solution is to batch the software pipelining into smaller groups of [2-4] work items at a time. For complex steps, my code may batch [1-8] work items in groups of four work items, resulting in up to two different instruction groups [1-4] and

[5-8], where each instruction group is software pipelined independently. This decreases the number of registers from 8× to 4×. For complex steps that also require range checks, my code may batch in groups of two work items, resulting in up to four instruction groups [1-2], [3-4], [5-6], and [7-8]. This decreases the number of registers from 8× to 2×. Although batching software pipelining in small groups of [2-4] work items decreases register pressure, it may also decrease ILP performance since the hardware scheduler now has fewer independent instructions to choose from.

Memory Reuse decreases Shared Memory Pressure: To save shared memory space, a single large memory buffer per thread block is used, which is partitioned into smaller per-warp and per-block buffers. Each per-warp buffer is then reused for multiple different purposes (loading keys, counting digits, scanning starts, extracting warp starts, etc.) during the various steps that make up my `BlockDistribute` method. The larger total memory buffer is used for sorting keys (and values) locally. In addition, there is a small per-block buffer used to store the S3 and S4 arrays. Since the twelve steps are sequential and do not overlap in time (for instance, scanning counts into starts does not begin until counting digits is completely finished), these two different uses of the same memory can safely reuse the same memory space for these different purposes. Practically speaking, I need to allocate each per-warp memory buffer as the largest of all these different warp arrays (*warpSpaceReq* = max(sizeArray(warp-keys), sizeArray(*thread-counts*), sizeArray(*thread-starts*), sizeArray(warp-values), …). I then need to allocate the entire memory buffer as the larger of either the aggregate of the warp arrays or the local sort buffers, also adding in the per block memory use (*spaceReq* = max(sizeArray(*nWarps***warpSpaceReq*, *block-keys*, *block-values*) + sizeArray(*S3* + *S4*)).

Memory Alignment: To enable the possibility of the compiler applying the Vector4 optimization to mitigate both load/store instructions and bank conflicts by a factor of 4×, I try to align the start of each block-array and warp-array in memory to a Vector4 boundary (16 bytes = 4*4) with extra pad columns where necessary.

9.4.3.2 `BlockDistribute` *method*:

My `BlockDistribute` method (see Figure 9.8) distributes an entire block of ‹*key*, *value*› pairs (or ‹*key*› singletons) into their final sorted runs. This method is invoked on each data block by the `GPU_DistributeKeys` kernel as it marches down each data row, block by block. This method basically performs a local counting sort on a single data block to locally sort keys (and values) into sorted runs. It then globally distributes the locally sorted keys (and values) into their final sorted run positions.

This is complex method, which and it took me a long time to get it working correctly, so I apologize that the overview may also be complex. The twelve steps of this method are organized into five broad processing groups: Count Keys, Scan Counts, Sort Keys, Sort Values, and Move to next Block. After the overview, each individual step is discussed in depth.

Group) ***Count Keys*****:** In this group, all keys are counted into thread-counts (16 counters per-thread histogram). In Step 1 (`LoadKeys`), each thread loads a short sequential run of ‹*nWork*=[1-8]› keys from the unsorted input array. In Step 2 (`ExtractDigits`), each thread extracts the current chosen digit from each key and stores the resulting [1-8] digits in a single compressed register. Finally in Step 3 (`CountDigits`), each thread bins its [1-8] digits into its matching count histogram of thread-counts. A single compressed register (called regStarts) of [1-8] key-starts is also created in Step 3. Each key-start is a collision-free starting offset for each key relative to the other keys assigned to this thread. These key-starts are needed to uniquely offset keys, when two or more keys from the same work run collide on the same digit value.

Group) **Scan *Counts*:** This group is the most complex, so I will explain the general problem needing to be solved before I overview the individual steps. To enable sort, the local starts (unique sorted offsets within the fixed-size data block) for each key (and value) are needed. The main point of generating these starts is that each thread can then safely store all its assigned keys into the sorted array without colliding with other threads concurrently storing data at the same time. To generate these local starts, a hierarchical set of three relative starts (*warp-starts*, across warps within a thread block; *thread-starts*, across threads within a warp; and *key-starts*, across keys within a thread) are built and then accumulated. These three relative starts are built using a hierarchy of scans from the count histograms. **Note:** The key-starts were already built by step 3 (in group 1), while the thread-starts and warp-starts will be built by steps 4 and 5 (in this group).

In Step 4 (`ScanThreads`), the entire thread warp scans assigned thread-counts into thread-starts. Each thread-start is a collision-free starting offset for sorted data runs relative to the other threads within the same warp. Step 4 also generates and stores a single histogram of sixteen warp-sums for use in the next step. In Step 5 (`ScanWarps`), eight active threads scan the warp-sums from Step 4 into warp-starts. Each warp-start is a collision-free starting offset for sorted data runs relative to the other warps within the same thread block. Step 5 also generates a final histogram of sixteen block-sums (total run counts) and sixteen block-starts (starting run offsets) across the entire data block. In Step 6.0 (`AccumulateStarts`), the local starts [1-8] are built by accumulating warp-starts (Step 5), thread-starts (Step 4), and key-starts (Step 3).

Group) **Sort *Keys*:** With local-starts now built, the keys can be safely distributed into sorted runs. In step 7 (`ShuffleKeys`), each thread shuffles (locally sorts) its assigned keys into sorted runs (kept in a local shared memory array). Local sorting increases overall performance by increasing run coherence, which reduces the number of "stores" required to distribute the keys into sorted runs in the next step. In Step 8 (`MapKeys`), each thread loads and maps (globally sorts) a short sequential run of [1-8] locally sorted keys into global memory. The global offsets are created by accumulating *row-starts* with relative run offsets from the start of each sorted local run. Each row-start is a collision-free starting offset for sorted data runs within the entire data set relative to other thread-blocks within the entire grid (CTA).

Group) **Sort *Values***: Optionally, the values also need to be distributed into sorted runs. If sorting ‹*key*, *value*› pairs instead of just ‹*key*› singletons, then the methods corresponding to Steps 9 – 11 need to be invoked. Otherwise, the `BlockDistribute` method can safely skip over to Step 12. In Step 9 (`LoadValues`), each thread loads a short sequential run of ‹*nWork*=[1-8]› values. In Step 10.0 (`ShuffleValues`), each thread shuffles (locally sorts) its assigned values (reusing the local starts from Step 6). In Step 11 (`MapValues`), each thread loads and maps (globally sorts) a short sequential run of [1-8] locally sorted values into global memory (reusing the global offsets from step 8).

Group) ***Move to Next Block*:** In step 12 (`UpdateRowStarts`), the thread block prepares to distribute the next data block along the current data row by adding the final block-sums (from Step 5) into the current row-starts. This safely skips past the just distributed keys (and values) for the current data block.

Now that a broad overview of the `BlockDistribute` method has been presented, next follows an in depth discussion of each of the twelve steps.

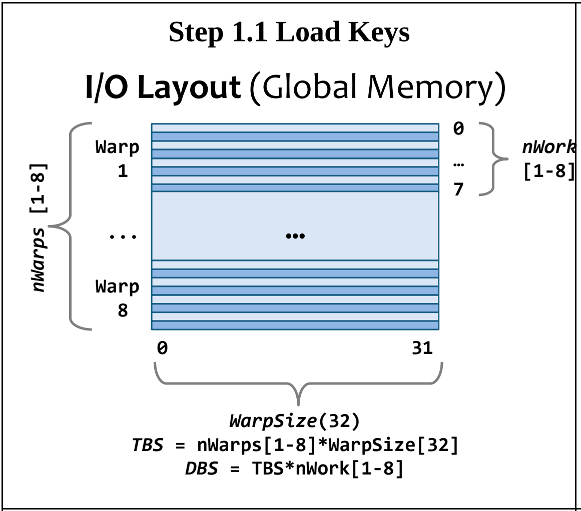
STEP 1) `LoadKeys` method:

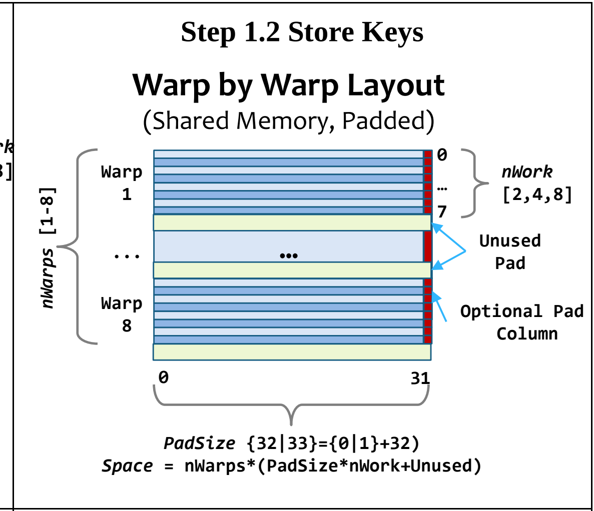
The `LoadKeys` method takes as input the *inKeys* parameter and produces as output a short run of ‹*nWork*=[1-8]› sequential keys stored in registers. For better performance, I reuse the 3-step conversion described in section 6.7.1 to convert between a warp-by-warp layout (that respects coalescence) kept in global memory into a sequential layout (required for correct results) kept in registers. These three sub-steps quickly cycle through three different views of memory (via three pointers) as summarized in Figure 9.9.

Sub-step 1.1) Each thread warp loads its own assigned section of [1-8] data warps of keys from the current data block in the *inKeys* array kept in global memory. These [1-8] keys are temporarily stored in registers.

Sub-step 1.2) Each thread warp stores its [1-8] keys as data warps into a conversion array kept in shared memory, each data warp may have an optional pad column to avoid bank conflicts using the "Pad and Rake" technique described in section 6.7.2.

Sub-step 1.3) Each individual thread within the thread warp loads a short sequential run of [1-8] keys from its assigned section of the conversion array kept in shared memory.





Step 1.3 Load Run (No Pad)

```
Load Run of 3 (Per Thread)
--------------------------------
01201201201201201201201201201201
20120120120120120120120120120120
12012012012012012012012012012012
................................
```

Step 1.3 Load Run (With Pad)

```
Load Run of 4 (Per Thread)
--------------------------------
*0123012301230123012301230123012
3*012301230123012301230123012301
23*01230123012301230123012301230
123*0123012301230123012301230123
0123............................
```

Figure 9.9: Four views of memory required to Load Keys efficiently. For Sub-step 1.1) the upper-left panel shows a warp by warp layout of the *inKeys* arrays as a single fixed-size data block, with each thread warp being assigned [1-8] sequential data warps. For Sub-step 1.2) the upper-right panel shows a warp by warp layout of the same data warps into shared memory with optional padding to help mitigate bank conflicts. For Sub-step 1.3) Both lower panels show how the short runs would be assigned to all 32 threads for a run of size 3 (without padding) and for a run of size 4 (with padding), where the extra pad column mitigates 4-way bank conflicts.

There are 3 main issues associated with this `LoadKeys` method – bank conflicts, converting between views, and range checks.

Mitigating Bank Conflicts: Bank conflicts caused by loading short sequential runs can be mitigated using the “Pad and Rake” technique (as described in section 6.7.2) via this kernel’s *IO_pad* template parameter, as shown in Table 9.7.

nWork	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
***k*-Way BC**	0	**2**	0	**4**	0	**2**	0	**8**
Extra Cycles	0	**1**	0	**3**	0	1	0	**7**
IO_pad	0	**1**	0	**1**	0	0*	0	**1**

Table 9.7: Bank Conflicts for increasing run length.

In this table, the *nWork* row shows the run length [1-8] to load; the *k-Way BC* row shows the expected number of bank conflicts per memory access [0,2,4,8]; the *Extra Cycles* row shows the cost in extra machine cycles per shared memory access; and the *IO_pad* row shows whether to turn on {1} or off {0} the “Pad and Rake” technique to avoid these *k*-way bank conflicts. Recall that the “Pad and Rake” technique only works for run lengths that are powers of two. So I only set *IO_pad* = {1} for ‹*nWork*=[2,4,8]›. In the outlier case of a run of length 6, it is cheaper to live with the resulting 2-way bank conflicts then to fix the problem.

Conversion between Views: For a stable Counting Sort, each thread needs to load a short sequential run of keys (stride =1). However, for best I/O performance via coalescence, keys should be loaded using the warp-by-warp BASk pattern (stride = 32). This problem was solved using the 3-step conversion process already described.

Range Checks Issue: My ROW DASk requires four similar versions of code when transferring I/O between global memory and registers, which supports the [‹FIRST?›, ‹MIDDLE*›, ‹LAST?›] range check pattern. Since this method loads keys from the *inKeys* array, four different range checked versions of this method are also required:

`LoadKeys_RC_BOTH` does [*start*, *stop*] range checks, which is typically only needed for a input range smaller than a single fixed-size data block (*n* ‹ *DBS*).

`LoadKeys_RC_START` does [*start*, …) range checks, which is typically only needed if the *start* offset is not warp aligned to a warp boundary [0, 32, 64, …].

`LoadKeys_RC_STOP` does (…, *stop*] range checks, which is only needed by the very last partially covered data block in the data set.

`LoadKeys_RC_NONE` does *NO* range checks, the vast majority of data blocks should end up calling the 'NONE' version, which is another reason to use my Row DASk.

In all four versions of `LoadKeys`, instructions are regrouped and reordered using software pipelining. This results in better ILP performance. The three range-checked `LoadKeys` functions support software pipelining using smaller batches of [2-4] keys at a time to reduce register pressure. The `LoadKeys_RC_NONE` function software pipelines all eight keys as a single batch of [1-8] work items.

STEP 2) `ExtractDigits` method:

The `ExtractDigits` method takes [1-8] keys as inputs and produces a single compressed 32-bit value containing [1-8] digits. Each chosen digit is extracted from its corresponding key by (optionally) transforming the key into a 32-bit unsigned integer and then shifting and masking off the chosen digit (4-bits) from this 32-bit key. The resulting 4-bit digits [1-8] are then compressed into a single 32-bit register called *regDigits*. The layout of 4-bit digits within this register is [8462 7351] instead of the more typical [8765 4321]. This specific layout pattern supports bit-level parallelism by allowing quads and pairs to be easily extracted using fewer shifts and masks. Also, type upscaling from 4-bits to 8-bits or 16-bits respectively basically costs nothing extra when extracting these quads and pairs. Instructions are software pipelined and reordered in batches of [2-4] keys to increase ILP performance. The `ExtractDigits` method is illustrated in Figure 9.10.

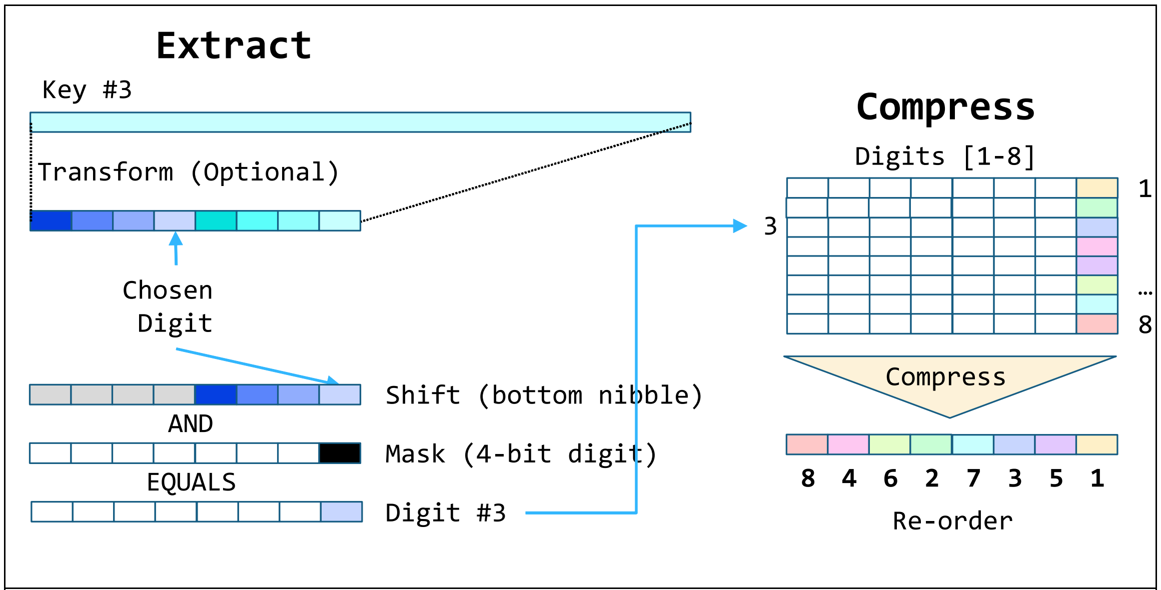


Figure 9.10 - Extract Digits from Keys: Each individual key is optionally transformed into a 32-bit integer. Then a chosen digit is extracted from the 32-bit key by shifting and masking. Then the resulting [1-8] digits are compressed and reordered into a single 32-bit integer with each 4-bit nibble representing a single digit. The reordered layout of 4-bit digits within the 32-bit register is [8462 7351].

STEP 3) `CountDigits` method:

The `CountDigits` method takes [1-8] compressed digits (stored in *regDigits*) and counts the digits into a per thread counts histogram containing 16 bins (one bin per possible digit value [0-15]). For each digit in *regDigits*, it extracts the current digit, bins the digit, and increments the corresponding counter in the histogram. The bins in each count histogram are represented by sixteen 8-bit counters, which are compressed and stored in four 32-bit lane counters in order to save shared memory space. Since each thread counts at most eight digits per counter, any overflow of 8-bit counters is not yet possible. Before counting any digits in any thread, all counts are reset to zero to ensure correct final counts in each histogram. Counting instructions are reordered using software pipelining in batches of [2-4] digits at a time to increase ILP.

The memory layout for `CountDigits` is shown in Figure 9.12. As can be seen in this figure, there are a total of four rows (one for each 32-bit lane) and 32 columns (one for each thread in the warp). Optionally, the Kernel's "S1_pad" template parameter can be used to turn on the "Pad and

Rake" technique for use in step 4.0 (`ScanWarps`). This adds a single pad column to the end of each lane row. The total number of 32-bit values required for counting in each individual warp is thus 128 (or 132 with pad).

There are two main issues with this counting step:

Counting Bottleneck Issue: Incrementing [1-8] per-thread counters cannot be safely software pipelined due to possible collisions between concurrently executing increment instructions. Consequently, the method must live with the stalls caused by the RAW dependencies between the load, increment, and store instructions associated with each count operation. This issue was previously described in Section 9.3.1.1.

Bank Conflict Issue: Unfortunately, there is a conflict between Step 3.0 `CountDigits` and Step 4.0 `ScanWarps`. To mitigate bank conflicts when loading and storing runs in Step 4, the "Pad and Rake" technique can add a single pad column per lane. This pad column impacts the data layout for both steps 3 and 4, as shown in Figure 9.12.

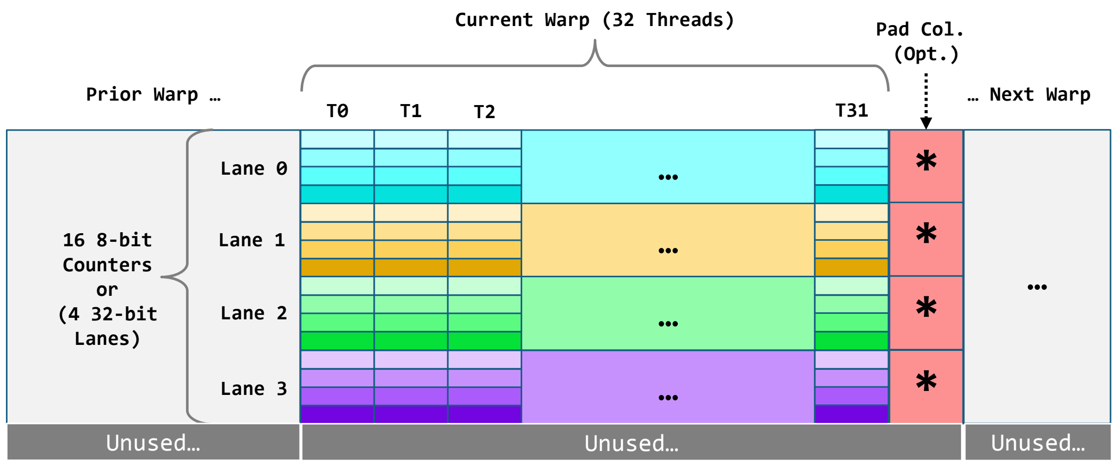


Figure 9.12 - `CountDigits` Memory Layout: Each thread in a thread warp has its own unique histogram of sixteen 8-bit counters compressed intofour4 32-bit lanes. Across all threads in a warp, there are four rows of 32 columns (+ optional pad column) resulting in a total of 128 (or 132) 32-bit lane values. Each thread warp is assigned its own unique memory array used for counting. Step 3 accesses per-thread count histograms in columns (vertically). However, Step 4 accesses short runs of 4 lane counters across each row (horizontally).

Adding the pad column can avoid bank conflicts when scanning runs in step 4. However, adding the pad column also negatively impacts step 3, which uses the same memory layout but with a different view of the data. Adding the pad column to the shared memory layout in Step 3 shifts each per-thread count histogram from being a straight vertical line on one bank column into a diagonal line spread over multiple bank columns. This shift results in an unpredictable number of bank conflicts on each shared memory access. Assuming a uniform random distribution of digits into counters results in a rough estimate[9] of 3.4 bank conflicts per shared memory access on average across the entire thread warp. Since there are up to sixteen shared memory accesses to count [8] digits, this approach results in up to 54.4 extra cycles due to bank conflicts in Step 3. On the other hand, not adding the pad column results in a guaranteed four-way bank conflict for each shared memory access in Step 4. Since there are a total of 25 shared memory accesses to scan all four data warps, **not** padding results in 100 extra cycles due to bank conflicts in Step 4. Since 54.4 < 100, I have found it best for overall performance to always set *S1_pad* = 1 to turn on the "Pad and Rake" technique in Step 4 even though it causes unavoidable bank conflicts in Step 3.

STEP 4) `ScanWarps` method:

The `ScanWarps` method exclusively scans thread-counts into thread-starts for a single thread warp. Recall that there are sixteen rows of 8-bit counters compressed into four 32-bit lanes. This results in a total of 128 32-bit values (132 with optional pad) that need to be scanned. Compressing four 8-bit counters per lane decreases work via bit-level parallelism (BLP), by allowing the code to scan four rows of 8-bit counters at once by scanning 32-bit lane values instead. Scanning each row of lane values also results in sixteen warp-sums (one per counter

[9] Given 4 rows, each thread in the warp can now potentially collide with 3 other threads on each shared memory access. Assuming a uniform random distribution of digits into counters, the chances of [1,2,3,4]-way collision for each thread are then [27/64, 27/64, 9/64, 1/64], respectively. Across all 32 threads in a warp, the chances of [1-4]-way conflicts become [1.01×10^{-12}, 0.22%, 60.29%, 39.59%] respectively, which leads to an average bank conflict of 3.39.

row), which are stored in the S3 block array as eight lanes per warp (2 16–bit sums per 32-bit lane value). These S3 warp-sums will finish being hierarchically scanned in Step 5 (`ScanBlocks`).

The memory layout for Step 4 is the same as in Step 3 (4 lanes × {32|33} columns). However, the pointer used to scan the data uses a row orientation instead of a column orientation (as in Step 3). The actual `ScanWarps` method uses a hierarchical set of scans on short runs and follows the scan-then-san parallel scan pattern (as described in Section 6.4.2). Individual runs are kept in registers to decrease shared memory accesses and to enable the Sklansky serial scan (as described in section 6.6.2). The method proceeds in three sub-steps, as shown in Figure 9.13: 1) **S1:SS‹4›**, 2) **S2:SS‹8›**, 3) **S1:SU‹4›**.

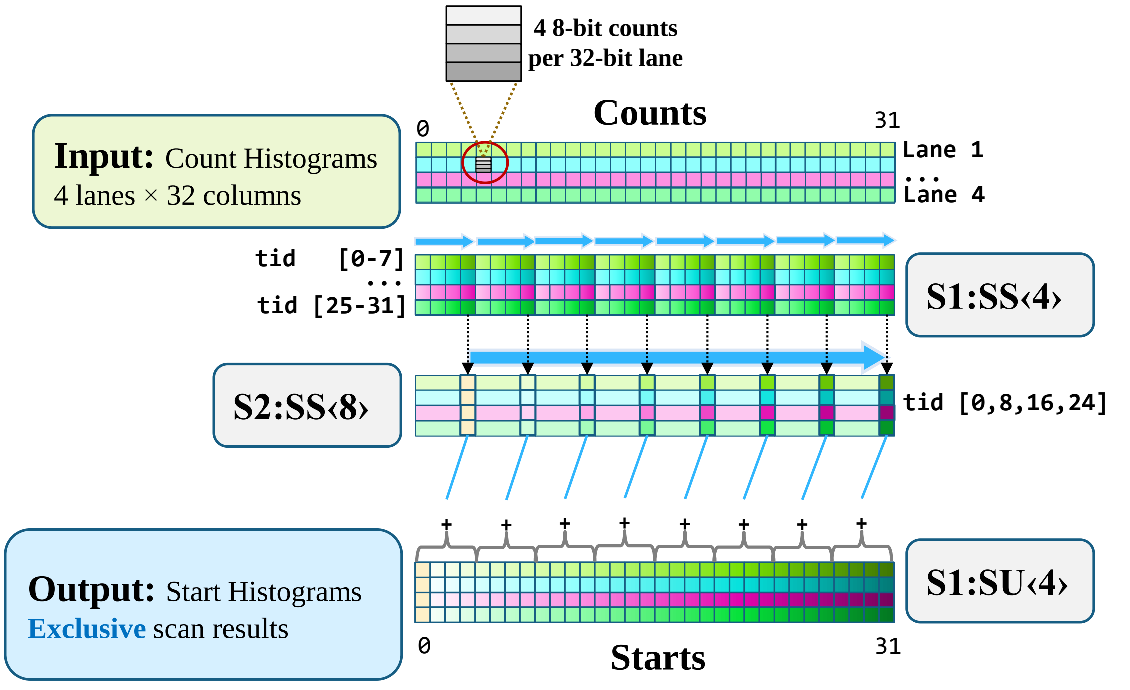


Figure 9.13 – `ScanWarps`: The `ScanWarps` method hierarchically scans warp-counts into warp-starts in 3 steps. In step 4.1 (**S1:SS‹4›**), each thread loads and scans a sequential run of 4 warp-counts. The resulting S1 run-sums are stored in the S2 array. In step 4.2 (**S2:SS‹8›**), each active thread [0,8,16,24] loads and scans a sequential run of 8 S1 run-sums. The resulting upscaled 16-bit S2 run-sums are stored in the S3 array. The *exclusive* S2 scan results are stored in the S2 array. In step 4.3 (**S1:SU‹4›**), each thread accumulates an S2 prefix into its S1 run. The *exclusive* S1 scan results are stored in the S1 array to generate the final warp-starts.

Sub-step 4.1 (S1:SS‹4›): Each thread in the warp loads a short sequential run of four counters (lane values) from the S1 array and then inclusively scans the run (in registers) using Sklansky's serial scan (see section 6.6.2). The resulting run-sum (*Sum*[1-4]) is stored back into the S1 array (which is also the S2 array) for further hierarchical scanning in sub-step 4.2.

Note: Since the entire "S1 run" is currently stored in registers, it is safe to reuse S1's memory for scanning the S2 array. Reusing memory saves space.

Sub-step 4.2 (S2:SS‹8›): Four active threads [0,8,16,24] in the current thread warp each load a sequential run of eight S1 run-sums from the S1 array and then they inclusively scan the run (in registers) using Sklansky's serial scan (see section 6.6.2). The resulting S2 run-sums are then upscaled and stored into the S3 array. Upscaling converts four 8-bit counters (stored in one 32-bit lane) into four 16-bit counters (stored in two 32-bit lanes) to avoid 8-bit overflow. The exclusive S2 scan results are then stored back into the S1 array to be used as prefixes in sub-step 4.3. Recall that an inclusive scanned run can be converted into an exclusive scanned run by reaching back one column and pre-pending the identity ($\mathbb{I}$). The identity is zero ($\emptyset$) for addition. Given an inclusive scanned run represented as [*s*1, *s*2, … , *s*8], the exclusive scanned run should be stored as [$\emptyset$, *s*1, … , *s*7].

Sub-step 4.3 (S2:SS‹8›): Each thread in the warp loads the correct S2 prefix from Sub-step 4.2 and accumulates that prefix into its current inclusive S1 run (stored in registers). The exclusive S1 scan results are then stored back into the S1 array as thread-starts for reuse in Step 6.2. Figure 9.13 shows the pseudo-code for all three steps (4.1-4.3).

```
Step 4.1: S1:SS‹4›
   All threads
...
Load S1 run
S1_1 = S1[0]
S1_2 = S1[1]
S1_3 = S1[2]
S1_4 = S1[3]
Scan S1 run Inc.
  // Stage 1
S1_2 = S1_1+S1_2
S1_4 = S1_3+S1_4
  // Stage 2
S1_3 = S1_2+S1_3
S1_4 = S1_2+S1_4
Store S1 run sum
S1[3] = S1_4
...
```

```
Step 4.2: S2:SS‹8›
  4 active threads
bActive = (0==tid%8)
if (bActive)
  Load S2 run
  //S1_4 = S1[3]
  S2_2 = S1[ 7]
  S2_3 = S1[11]
  ...
  S2_8 = S1[31]
  Scan S2 run Inc.
   // Stage 1
  S2_2 = S1_4+S2_2
  S2_4 = S2_3+S2_4
  S2_6 = S2_5+S2_6
  S2_8 = S2_7+S2_8
   // Stage 2
  S2_3 = S2_2+S2_3
  S2_4 = S2_2+S2_4
  S2_7 = S2_6+S2_7
  S2_8 = S2_6+S2_8
```

```
Step 4.2: Cont…
   // Stage 3
  S2_5 = S2_4+S2_5
  S2_6 = S2_4+S2_6
  S2_7 = S2_4+S2_7
  S2_8 = S2_4+S2_8
  Upscale
  S8_12=… // 8[1-2]
  S8_34=… // 8[3-4]
  Store S2 run sums
  S3[…] = S8_12
  S3[…] = S8_34
  Store S2 Exclusive
  S1[ 3] = 0
  S1[ 7] = S1_4
  S1[11] = S2_2
  ...
  S1[27] = S2_6
  S1[31] = S2_7
end (bActive)
```

```
Step 4.3: S1:SU‹4›
   All threads
Load S2 prefix
pre = S1[3]
Update S1 run
S1_1 = pre+S1_1
S1_2 = pre+S1_2
S1_3 = pre+S1_3
Store S1 Exclusive
S1[0] = 0
S1[1] = S1_1
S1[2] = S1_2
S1[3] = S1_3
```

Figure 9.13: `ScanWarp` code: Step 4.1 (*S1 scan*) inclusively scans a run of 4 counters. Step 4.2 (*S2 scan*) inclusively scans a run of 8 S1 run-sums then stores the *exclusive* scan results. Step 4.3 (*S1 update*) adds the matching S2 prefix into each inclusive S1 run (from step 4.1) and then stores the *exclusive* scan results.

Performance Analysis: My `ScanWarps` code scans four data warps of warp-counts into four data warps of warp-starts using a single thread warp (32 threads). This code also generates as output the warp-sums required for Step 5.0 (`ScanBlocks`) to generate block-starts. The code was carefully written[10] to take advantage of ILP via software pipelining for increased performance.

The code was also purposefully written to overlap the S1 and S2 arrays in memory. This overlap was enabled by storing both the S1 and S2 runs in registers. My 32-element hierarchical scan provides five advantages:

- It enables reordering the load and scan operations to increase performance via ILP.
- It enables Sklansky's serial scan (in registers) for scanning both the S1 and S2 runs.
- It overlaps the S1 and S2 arrays in memory saving space (The S2 array size would otherwise require space equal to ¼ the size of the S1 array).
- It saves one register pointer (by reusing the S1 array pointer instead of creating an S2 pointer) and also saves the cost in instructions to setup the S2 pointer.

[10] I tried many different ways to scan four data warps using a single thread warp. This hierarchical method (S1:SS‹4›, SS‹8›, S3:SU‹4›) was the fastest (taking about 315 machine cycles). Compare this with an interleaved `WarpScan` (in shared memory) (WS‹32›) over four data warps (taking about 503 machine cycles). A modified interleaved `WarpScan` (in registers, .PTX shuffle) (WS‹32›) should be even faster (taking about 280 machine cycles) but only works on the GTX Titan or newer GPU architectures.

- It turns out by construction that the final S1 run-sum (S1_4) and the first entry in the S2 run (S2_1) are one and the same value. This insight saves one load instruction and one register in the final code.

My scan also has five disadvantages:

- A branch test, *bActive* = ((0==(*tid*%8)), is required to determine the four threads that are active during Sub-step 4.2 (S2:SS‹8›). The CUDA compiler generates three PTX instructions with RAW dependencies between each other, slowing ILP performance.
- Loading the S1 and S2 runs sequentially can result in four-way bank conflicts on (25 out of 27) shared memory accesses for a loss of at least 100 machine cycles to bank conflicts. Bank conflicts can be mitigated by using the "Pad & Rake" technique by turning on the (*S1_pad*=1) template parameter. Unfortunately, as already explained, avoiding bank conflicts in step 4.0 causes unavoidable bank conflicts in step 3.0.
- Storing both the S1 and S2 runs in registers increases register pressure. At least thirteen registers are required: two for the 64-bit S1 run pointer, four for the S1 run, seven (=8-1) for the S2 run, and two for upscaling from 8-bit to 16-bit.
- The CUDA LLVM compiler de-optimizes my scan code by reordering the loads and adding and using more registers than actually required. The only work-around I have found so far is to carefully rewrite the code in .PTX assembly. Unfortunately, this makes the resulting code much less general.
- There is a potential overflow issue on the final summation in the S2 scan (in Sub-step 4.2). How I deal with this is discussed next.

Overflow Issue: Recall that to save space and scan work, my code compressed sixteen 8-bit counters into four 32-bit lane counters. This means that overflow is now a distinct possibility. Although unlikely, if *nWork* =8, then all eight digits for all 32 threads could get binned into the same counter, with a maximum count of 256 (=8·32). This maximum count would overflow an 8-bit counter. However, if *nWork* = [1-7] then overflow cannot happen (7*32 < 255).

However, in step 5.0 (`ScanBlocks`), overflow becomes very likely. Consequently, the code handles overflow by upscaling four 8-bit counters stored in one 32-bit lane counter into four 16-bit counters stored as two 32-bit lane counters. The upscaled pair of lane values (four 16-bit counters) are stored in the S3 array. If overflow is not a possibility (*nWork* = [1-7]), then the code adds first and then upscales second before storing the final results. If overflow is a possibility (*nWork* = 8), then the code upscales the 8-bit counters into 16-bit counters first and then perform sthe final add before storing the final results in the S3 array. The two different approaches to handle overflow by upscaling are shown in Figure 9.14.

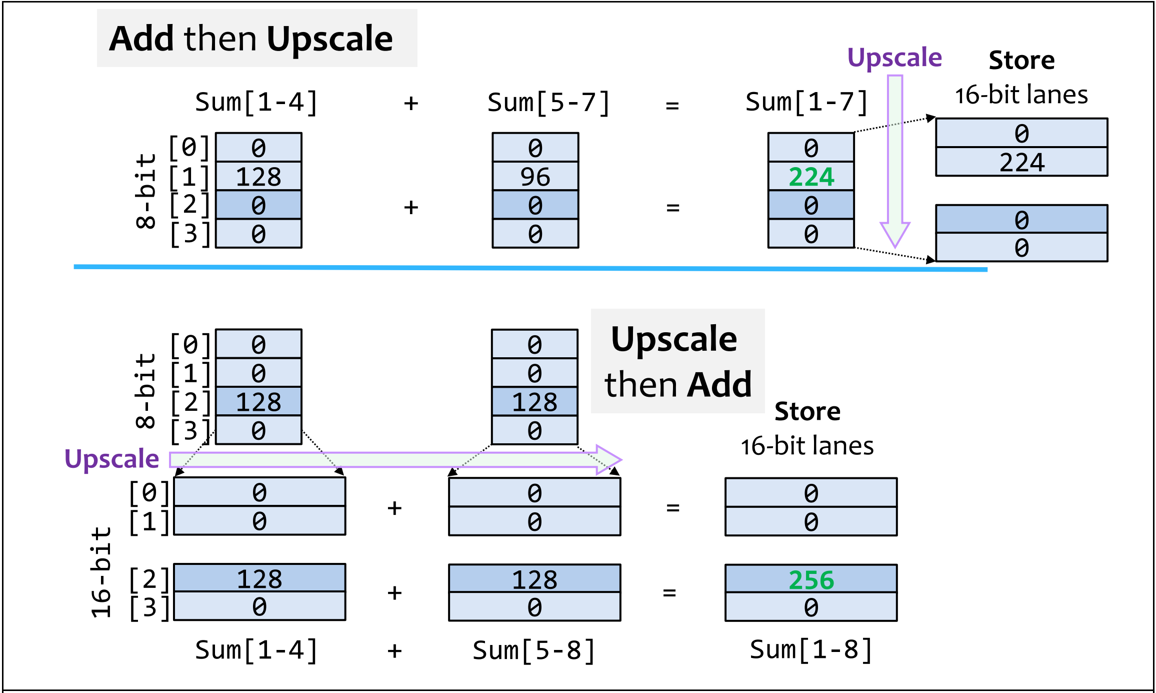


Figure 9.14 - Handle overflow by upscaling: In the top graphic, overflow is not possible, so the code adds first, upscales second, and then stores the final 16-bit counters. In the bottom graphic, overflow is possible, so the code upscales first, then adds second, and then stores the final 16-bit counters.

STEP 5) `ScanBlocks` method:

The `ScanBlocks` method exclusively scans warp-sums into block-starts using eight active threads. Recall that there are ‹*nWarps*=[1-8]› thread warps as specified by the kernel template parameter. During Step 4.0, `ScanWarps` scans the thread counts into thread starts and stores a complete histogram of sixteen upscaled warp-sums in the S3 array, where each warp-sum is the sum of all thread counts along each row of 32 threads. For better performance, the code scanned using stored four 8-bit counters per 32-bit lane. To avoid potential overflow, the code upscaled from four 8-bit counters to four 16-bit counters just before storing the sums into the S3 array. Thus, the sixteen warp-sums are stored as eight rows of lane values with two 16-bit counters per 32-bit lane value. Each individual thread warp generates its own per-warp count histogram stored as a column of eight lane values. As a result, the size of the resulting S3 array is

8**nWarps*([1-8]). This S3 array will be scanned into a final per-block count histogram of sixteen block-sums, which will be stored in the S4 array.

The S4 array layout (as shown in Figure 9.16) has a fixed array size with four different sub-arrays that are used for different purposes:

- **Row-starts:** The row-starts sub-array contains sixteen row-starts, which track the current start of each globally sorted array within the output.
- **Block-starts:** The block-starts array tracks the start of each locally sorted run within the current data block. This array includes an extra hard-coded "zero" value that is used for grabbing exclusive results by reaching back one column. Consequently, this array contains seventeen values (one zero + sixteen inclusively scanned block-starts).
- **Block-start-pairs:** The block-start-pairs array is used to scan the final block-sums into final block-starts. The 16 block-sums are compressed into 8 lane counters (2 16-bit counters per lane). The first four elements of this array are hard-coded to zero to eliminate branching during parallel `WarpScan` (See Section 6.6.3). Consequently, this array contains 12 values (4 zeros + 8 compressed lane counters).
- **Block-sum-pairs:** The block-sum-pairs array contains eight elements, representing the compressed final block-sums for the entire data block. This array is used in step 12.0 to update the row-starts to skip past the just sorted current data block.

An additional unused three pad elements are added to the end of the S4 array. Consequently, the final array size is 56 elements, which is a multiple of eight. The memory layout for the S3 and S4 arrays are shown in Figure 9.15.

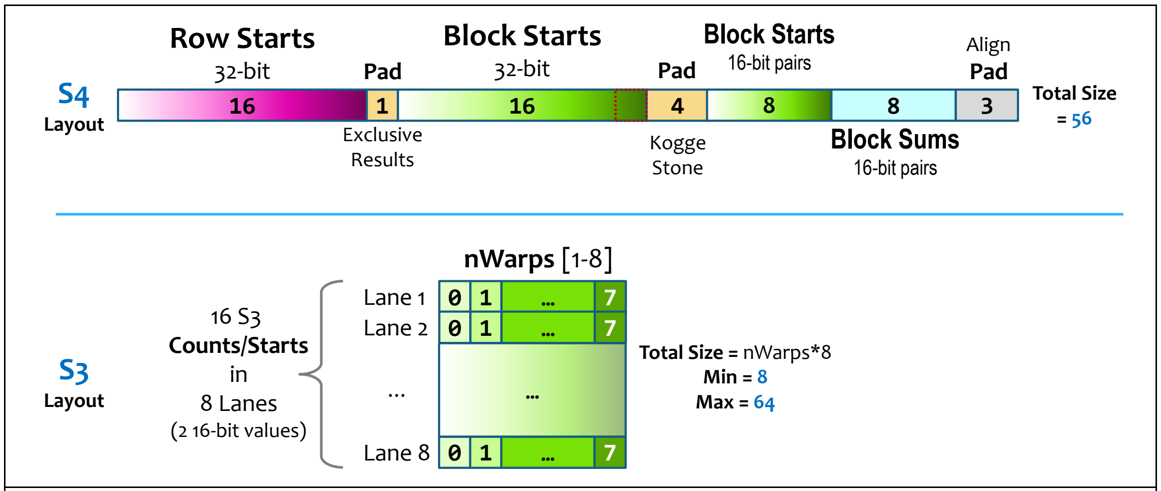


Figure 9.15 - `ScanBlocks` Memory Layout. The S4 array contains 16 *row-starts*, 17 *block-starts* (+1 pad column for exclusive results), 12 compressed *block-start-pairs* (+4 pad columns for parallel `WarpScan`) and 8 compressed *block-sum-pairs*. Finally, 3 unused pad columns are used to align the S4 array to an 8-element boundary, resulting in 56 (32-bit) elements in total. The S3 array contains 16 warp-sums per thread warp, stored as 8 rows of compressed lane values (2 16-bit counters per lane) with one column of warp-sums/block-starts per thread warp (*nWarps* = [1-8]). The S3 array is thus already aligned to an 8-element boundary with a minimum and maximum size of 8 (8*1) and 64 (8*8) elements, respectively.

Because the overall scan is hierarchical, the warp-starts generated in the `ScanWarps` method are missing the correct prefixes for the thread-warps (and their counters). These missing prefixes represent the total sum of all counters, which would normally precede the current warp-starts (as in a serial sequential scan). The `ScanBlocks` method provides these missing prefixes by scanning the warp-sums into block-starts. The actual `ScanWarps` method, as shown in Figure 9.16, proceeds in three main sub-steps: S3:SS‹*nWarps*›, S4:WS‹8›, and S3:SU‹8›.

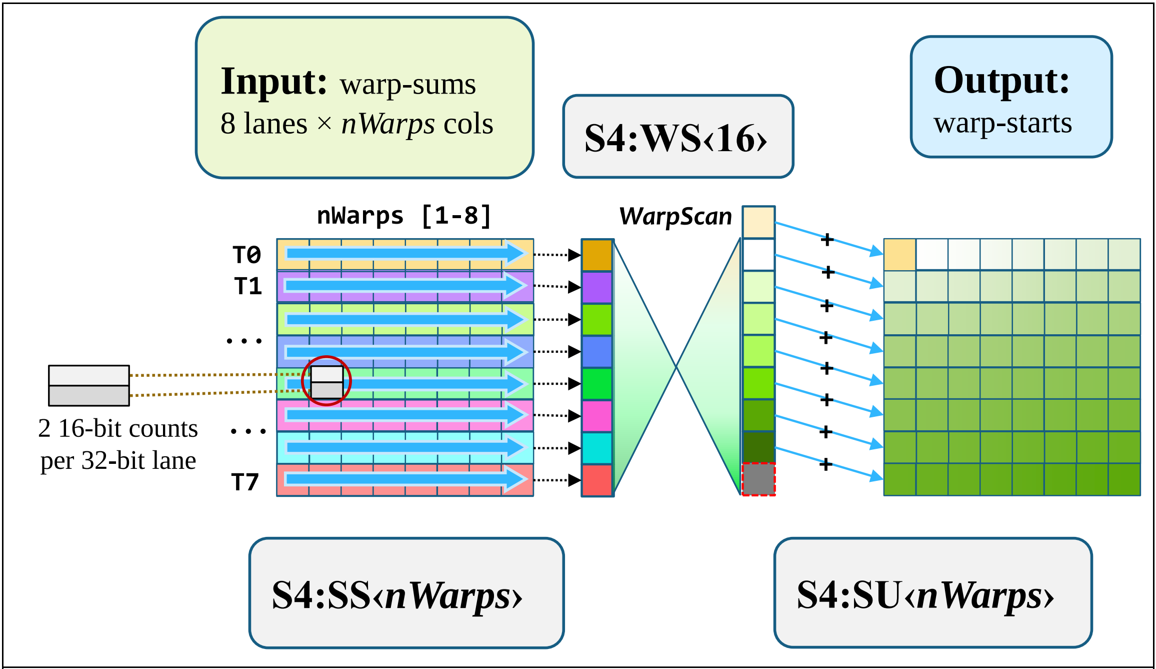


Figure 9.16: The `ScanBlocks` method scans warp-sums into warp-starts in three steps. In step 5.1 (**S3:SS‹*nWarps*›**), 8 active threads ([0-7]), each loads and scans a short sequential run of ‹*nWarps*=[1-8]› warp-sums. The resulting S3 run-sums are stored into both the S4 Block-Sum-Pairs and Block-Start-Pairs arrays. In step 5.2 (**S4:WS‹8›**), 8 active threads ([0-7]) cooperatively `WarpScan` the 16 block-sums into 16 block-starts. The resulting inclusive pairs are decompressed and stored in the S4 block-starts array and then reloaded and recompressed as an exclusive scanned run. In step 5.3 (**S3:SU‹*nWarps*›**), each thread adds an S4 prefix into its S3 run. The *exclusive* S3 scan results are stored in the S3 array to generate the final warp-starts.

Sub-step 5.1 (S3:SS‹*nWarps*›): Eight active threads [0-7] in the first warp each loads a short sequential run of ‹*nWarps*=[1-8]› warp-sums (lane values) from the S3 array and then inclusively scans the run in registers. The resulting block-sum (Sum[1-*nWarps*]) is stored into both the S4 block-sum-pairs array (for use in Step 12.0) and the S4 block-start-pairs array (to be scanned in sub-step 5.2).

Sub-step 5.2 (S4:WS‹8›): Eight active threads [0-7] in the first warp cooperate to `WarpScan` the elements stored in the S4 block-start-pairs array from block sums into block starts. The parallel `WarpScan` method generates an inclusive run of block-starts, but exclusive results are needed in Step 5.3. To unpack/repack the compressed pairs properly, the code uses a 5-step solution:

Step 1) It decompresses the eight scanned lane values into sixteen block starts.

Step 2) It stores the sixteen block starts in the S4 block-starts array.

Step 3) It has each thread load an exclusive pair of two block-starts by reaching back one column in the S4 block-starts array.

Step 4) It compresses the sixteen exclusive pairs into eight lane values.

Step 5) It stores the compressed lane values back into the S4 block-start-pairs array for use as prefixes in Sub-step 5.3.

Sub-step 5.3 (S3:SU‹8›): Eight active threads [0-7] in the first warp each load the correct prefix from sub-step 5.3 (in the S4 block-start-pairs array) and adds that prefix into its current S3 run (stored in registers). The exclusive S3 scan results are then stored back into the S3 array for use as warp-start prefixes in Sub-step 6.3.

Issues: There are two main problems with this step: Barrier synchronization and Idle threads

Barrier Issue: All warp-sums for each thread warp must be generated before this step can safely scan those warp-sums into block-starts. One easy solution is to insert a barrier (`syncthreads`) before and after this step. These barriers are necessary[11] for correct parallel behavior between multiple warps within a thread block.

Idle Threads Issue: Only eight threads are active in the first warp during the entire execution of the `ScanBlocks` method. This means the other 24 threads in the first thread warp do no useful work during this time and the other [2-8] thread warps are inactive until all thread warps reach the second barrier.

STEP 6) `AccumulateLocalStarts` method:

The `AccumulateLocalStarts` method generates the local starts needed to correctly sort each key (and value). These local starts allow each thread to safely store all its assigned keys into a local keys array (in shared memory) without colliding with other parallel threads concurrently storing data. Recall that the local starts are built by accumulating: warp-starts, thread-starts, and key-starts, as depicted in Figure 9.17.

[11] Note: a thread block containing only a single warp does not need barriers for correct parallel behavior, as all threads within a thread warp move in lock-step through the code.

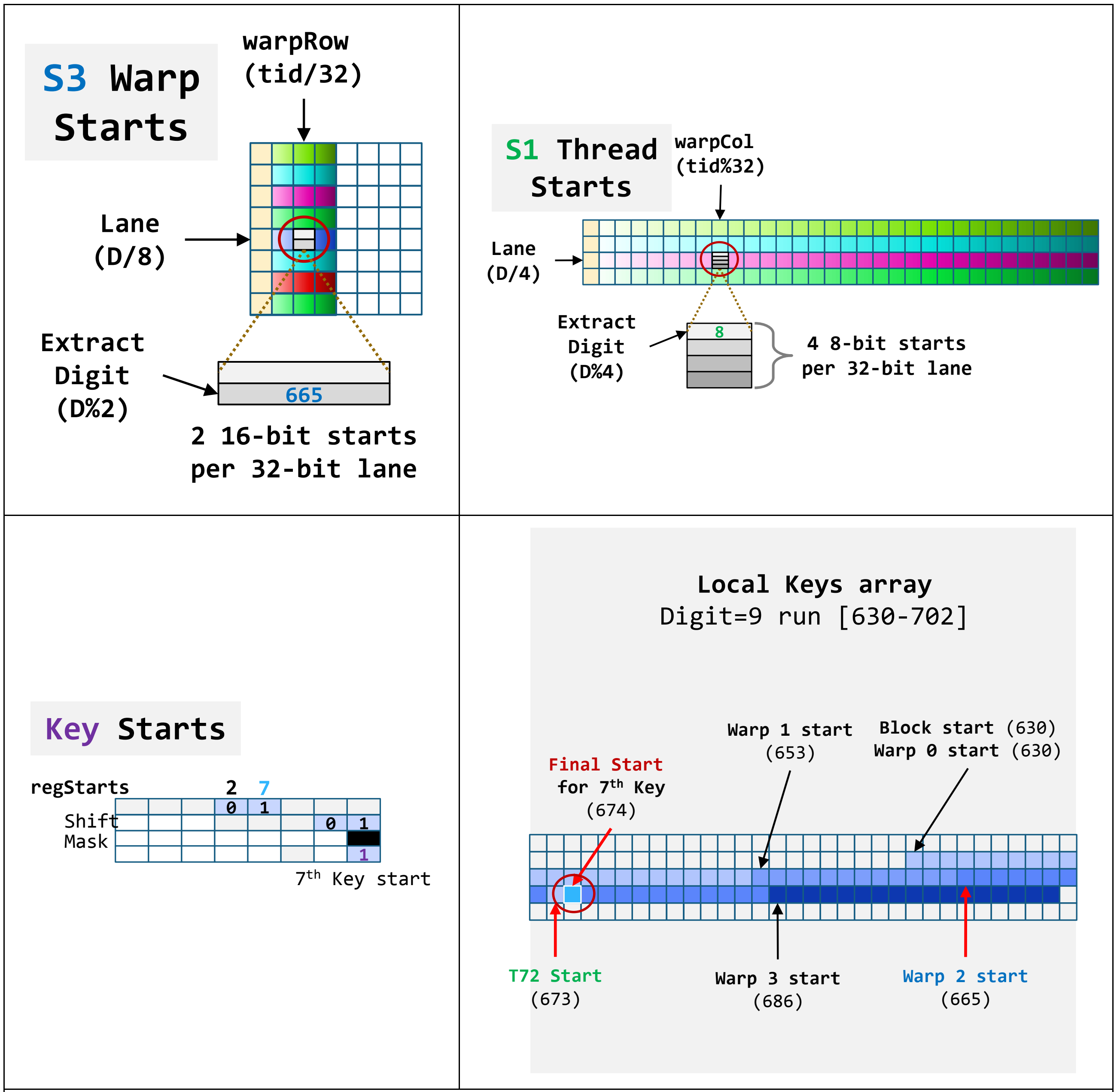


Figure 9.17: Local Start = Warp-Start + Thread-Start + Key-Start. The original start for each key was split and generated hierarchically in Steps 3, 4, and 5. The local start is the accumulation of the other 3 starts. *Key-starts* (Lower-Left panel) are extracted from a compressed "regStarts" register using shift & mask operations. Thread-starts (Upper-Right panel) are extracted from the S1 array using the thread id and digit to identify the row ([0-3]) and column ([0-31]), respectively. Warp-Starts (Upper-left panel) are extracted from the S3 array using the thread id and digit to identify the row [0-7] and column [0-*nWarps*-1]. One example (Lower-Right panel) of how one run (digit = 9) is divide into warp-starts, thread-starts, and the final start is shown. The example assumes 4 thread warps per block, threadId = 72, and the 7th key in a run of 8 keys.

- **Warp-starts:** The warp-starts (Figure 9.17, upper-left panel, stored in the S3 array and built in Step 5.3) provide the missing prefix sums from preceding warps within the same thread block.

- **Thread-starts:** The thread-starts (Figure 9.17, upper-right panel, stored in the S1 array and built in Step 4.3) provide the missing prefix sums from preceding threads within the same warp.
- **Key-starts:** The key-starts (Figure 9.17, lower-left panel, stored in the compressed regStarts register and built in Step 3) provide the correct prefix-sums for the ‹*nWork*=[1-8]› sequential keys assigned to each thread (in case 2 or more keys share the same digit value).

The local start for each key is the sum of the corresponding warp-start, thread-start, and key-start for the unique ‹*warp* [0-7], *thread* [0-31], *key* [0-7]› triplet.

In the lower-right panel of Figure 9.18, I show how a local start is built for a specific triplet for a specific data block, with ‹*warp, thread, key*› = ‹2,8,7›. Note that the actual key has a chosen digit value of 9 in this example. The local data block stores 1,024 keys. The block-start shows the starting offset (630) of the entire 9th digit value's run. The warp-start (+665) shows the starting offset (665) for the 2nd warps sub-run within the 9th digit run. The thread-start (+8) shows the starting offset (673 = 665+8) for the threads sub-run within the warps run. This run belongs to the 72nd thread in the thread block (which is also the 8th thread within the 2nd warp). Finally, since the 72nd thread had two keys that collided on the 9th digit value, the key-start (+1) shifts the 7th key by one to account for that prior key in the same work run, resulting in the final local-start value of 674 (= 665+8+1).

This method proceeds in three main sub-steps: Extract Key-starts, Accumulate Thread-Starts, Accumulate Warp-Starts:

Sub-step 6.1) Extract Key-Starts: Each thread extracts ‹*nWork*=[1-8]› key-starts from the compressed "regStarts" register, which was created in Step 3.0. This is done using simple bit-logic "shift & mask."

Sub-step 6.2) Accumulate Thread-Starts: For each of *i*= [1-8] keys assigned to the thread, the thread extracts the correct thread-start from the S1 array. The *i*th thread-start is then accumulated into the *i*th key-start. Since the thread-start is a prefix of the key-start, the addition is done as *keyStart* = *threadStart* + *keyStart*. The memory layout is shown in Figure 9.18 (upper-right panel), with four compressed lanes in four rows and 32 columns (one per thread in the warp). Each 32-bit lane value contains four 8-bit thread starts. So, additional work (shift & mask) must be done to extract the correct 8-bit start from each lane. This method basically works in six steps:

Step 1) It extracts the i^{th} 4-bit digit from the compressed *regDigits* register.

Step 2) It computes the lane offset of the correct lane value in the S1 array (*warpCol* = *tid*%32; *laneRow* = *digit*/4; *laneOff* = (*laneRow***S1_rowSize*)+*warpCol*). Note: *S1_rowSize* is either {32|33} depending on whether the "Pad and Rake" technique is turned on using the *S1_pad* template parameter.

Step 3) It computes the shift and mask values (for [0-3] = *digit*%4) required to extract the correct 8-bit start from the 32-bit lane value.Step 4) Load the lane value from the S1 array (using the lane offset from step 2).

Step 5) It cxtracts the thread-start from the lane value (using the shift and mask from step 3).

Step 6) It accumulates the thread-start into the key-start (*keyStart* = *threadstart* + *keyStart*).

The instructions for extracting S1 thread-starts are grouped and reordered using software pipelining in batches of [2-4] starts at a time to increase ILP. As already discussed in Step 3.0, this sub-step can cause bank conflicts when using the "Pad & Rake" technique, which can result in an unpredictable number of bank conflicts, typically costing ~3.4 extra machine cycles per shared memory access.

Sub-step 6.3) Accumulate Warp-Starts: For each of the *i* = [1-8] keys assigned to the thread, the thread extracts the correct warp-start from the S3 array. The S3 memory layout is shown in figure 9.18 (upper-left panel), with eight compressed lanes in eight rows and ‹*nWarp*=[1-8]› columns (one per warp in the thread-block). Each 32-bit lane value contains two 16-bit thread starts. So, additional work (shift & mask) must be done to extract the correct 16-bit start from each lane. This method basically follows the same six main steps as sub-step 6.2 with some minor differences to account for the different memory layout. There is a minor issue with bank conflicts in this sub-step; however, because of the small size of the S3 array ([8-64] elements), there will be at most 2-way bank conflicts on each access.

STEP 7) `ShuffleKeys` (Local Sort) method:

The `ShuffleKeys` method distributes keys from the current data block into sorted runs into a local array kept in shared memory. The implementation of this method is straightforward. The [1-8] original keys from Step 1.0 (`LoadKeys`) are stored directly into the local array at the positions indicated by the corresponding [1-8] local starts from Step 6. Since there are sixteen possible values for each digit [0-15] extracted from each key, the sorted keys (or values) ends up sorted into sixteen contiguous runs. The relative order of input keys (within the current data block) with the same digit value [0-15] is preserved in the final order of each sorted digit run. Thus, this local sort is stable. This stability is a direct result of the code carefully preserving sequential sequences during loads and scans.

There are two main issues associated with sorting keys (and values): bank conflicts and efficient use of coalescence.

Bank Conflict Issue: Shuffling keys into shared memory is unpredictable as it is unknown ahead of time which keys will end up in which memory banks. As a result, there will be an unpredictable number of bank conflicts within each data warp being sorted. This is a classic "balls into bins" problem: assume a uniform random distribution of keys across memory banks., The problem is to find the expected maximum keys per bank (or in other words the maximum height of any bin). My own simulations on 100 thousand runs (32 balls into 32 bins) show the average bin height (expected number of bank conflicts) is about four (4-way bank conflicts on average), which would cost an extra three machine cycles per memory access on average. All the workaround solutions I have tried require many instructions and therefore decrease performance further. Therefore, in this case the best solution is to just live with the resulting bank conflicts.

Coalesced Memory Efficiency Issue: This local sort is clearly optional, so why do it? There is enough information to build global starts to distribute keys into sorted runs immediately available after step 6.0. So, why not just output each key to its final sorted position in the output array? The answer (as demonstrated by Dr. Ha's shuffle and map technique) is coherence. Coherence impacts output performance. As Figure 9.18 illustrates, unsorted keys have low coherence and require many uncoalesced stores to write into multiple output runs; sorted keys have high coherence and require fewer coalesced stores to write into the same runs.

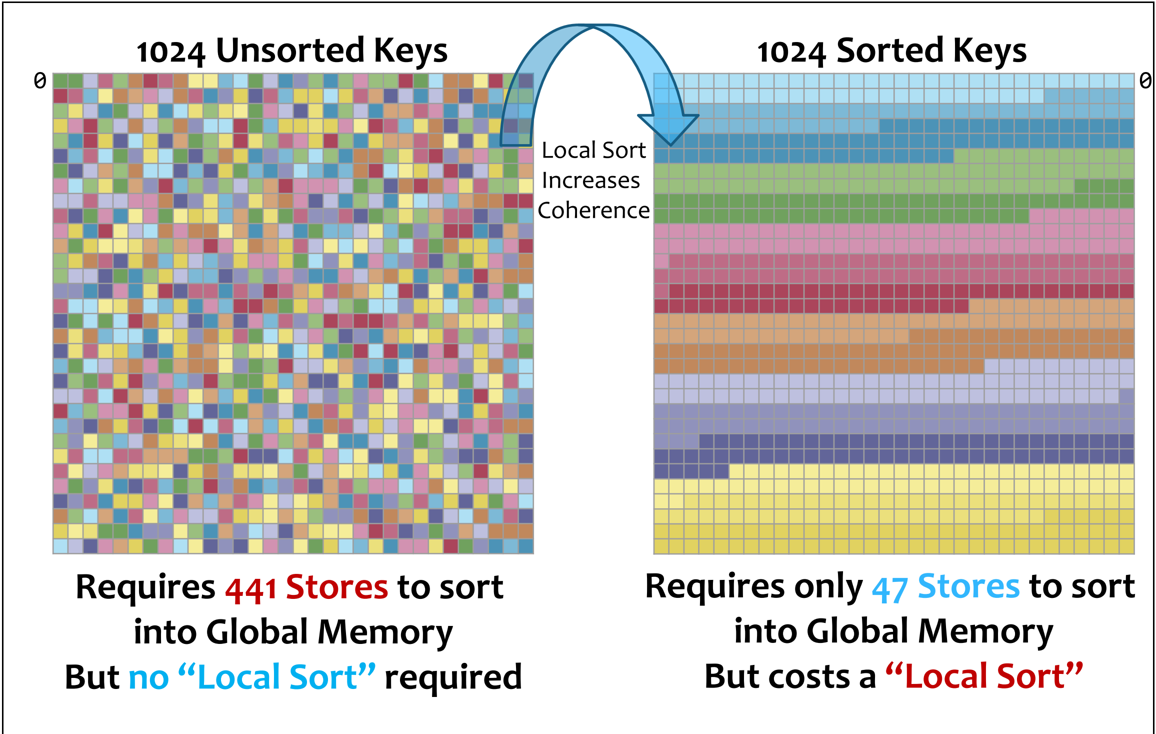

1024 Unsorted Keys
1024 Sorted Keys
0
0
Local Sort Increases Coherence
Requires 441 Stores to sort into Global Memory
But no "Local Sort" required
Requires only 47 Stores to sort into Global Memory
But costs a "Local Sort"

Figure 9:18 - `ShuffleKeys` (Local Sort): 1024 keys (32 rows × 32 columns) represented by their digit values [0-15] as 16 different colors. The left grid represents 1024 unsorted input keys assuming a uniform random distribution. The right grid shows the same keys locally sorted into 16 digit runs. Sorting the left grid (unsorted) directly into global memory requires 441 stores total (512 maximum in the worst case). However, the right grid (locally sorted) requires at most 47 stores (32 warps + 15 transitions) to sort into global memory.

To help measure how coherence impacts coalescence, I created a metric called *Coalesced I/O efficiency*, which is a percentage measure of the ratio of the average data elements transferred versus the maximum possible elements transferred (32) given a coherent run of data containing *n* elements. "Coherent" in this case means that all the data within this run is being written to the same output array. One easy way to achieve 100% coalesced I/O efficiency is to warp-align data transfers and to access data sequentially within a thread warp (meaning that the k^{th} thread within a warp accesses the k^{th} data offset within the data warp, $k \in [0\text{-}31]$). This is exactly how Step 1.0 loads keys from global memory into registers.

If the initial data start is not warp-aligned but is random or unpredictable, then the length of the data run helps determine the coalesced I/O efficiency. Typically, longer sequences of elements have larger aggregate increases in I/O throughput. Coalesced I/O efficiency ranges from a minimum of 3.125% (only one data element transferred per I/O operation) up to a maximum of 100.0% (32 data elements transferred per I/O operation). I/O sequences that are short or unaligned can only partially take advantage of coalescence. On the other hand, I/O sequences that are long and warp-aligned can fully take advantage of coalescence.

The following table (shown in Table 9.8) shows what happens to coalesced I/O efficiency as the data block size (DBS), alternately known as keys per block, is increased from 32 to 2048 in powers of two.

DBS	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**∞**
Avg. Run Length	2	4	8	16	32	64	128	n/a
Max. Stores	16	17	19	23	31	47	79	n/a
Avg. Keys per Store	2.00	3.76	6.74	11.13	16.52	21.79	25.92	32
Coalesced I/O Efficiency	6.25%	11.75%	21.00%	34.78	51.63%	68.09%	81.00%	100.00%

Table 9.8: Coalesced I/O efficiency as the DBS increases in size from 32 to 2048 in powers of two.

For this specific example of 4-bit radix sort, the maximum number of stores (*Max. Stores*) can be computed from the DBS as (DBS/32)+15, since there is at least one store required per data warp (32) and a maximum of 15 transitions (16 data runs – 1) across the entire sorted data block. The *Average Keys per Store* is computed by dividing the DBS by the maximum stores. Finally, the coalesced I/O efficiency is computed by dividing the Average Keys per Store by the maximum coalesced transfer rate (32 keys per store). As can be seen from the table, coalesced output becomes more efficient as the average run length increases, which in turn depends on the DBS. Therefore, throughput performance can be improved by increasing the work per thread and warps per block via the ‹*nWork*, *nWarps*› template parameters.

STEP 8) `MapKeys` (Global Sort) method:

The `MapKeys` method distributes keys into sorted runs in the output array in global memory. This method's implementation is straightforward and takes 4-5 sub-steps:

> **Sub-step 1)** Each thread loads a short sequential run of ‹*nWork*=[1-8]› keys from the sorted local array of keys in shared memory (from Step 7).
>
> **Sub-step 2)** The chosen digits [0-7] are extracted from the keys using the same transform, shift, and mask process as described in Step 2 (`ExtractDigits`).
>
> **Sub-step 3)** A local index (LI[0-7]) is computed from the keys. The local index is computed as *LI*[0-7] = (*tid***nWork*)+[0-7], where [0-7] is the current key [1-8]-1 being processed.
>
> **Sub-step 4)** For each key, a global index (GI) is computed. Optionally, a range check may also need to be applied to each global index to avoid writing keys that are out of bounds. When range checks are not required, sub-steps 3 and 4 are combined into a single sub-step. The global index is computed as *GI*[0-7] = *LI*[0-7] – *blockStarts*[digits[0-7]] + *rowStarts*[digits[0-7]]. The *blockStarts* array indicates the start of each of the sixteen sorted runs in the local array. The *rowStarts* array indicates the start of each of the sixteen sorted runs in the global array for the current thread block.
>
> **Sub-step 5)** Each thread outputs its run of keys [1-8] to global memory at the specified global index (GI[0-7]).

Within this method, instructions are software pipelined and reordered in batches of [2-4] work-items at a time to increase ILP performance. After this method completes, the entire current fixed-size data block of keys has been distributed to their unique sorted locations as sixteen digit runs in global memory.

If the radix sort is sorting ‹*key*, *value*› pairs instead of ‹*key*› singletons, then optional steps 9.0-11.0 also need to be invoked to sort the fixed-size data block of values corresponding to the just sorted block of keys. Otherwise, the algorithm can safely skip to Step 12.0.

There is are two main issues: bank conflicts and range checks.

Bank Conflict Issue: I did not add code to "Pad & Rake" when loading the runs to mitigate bank conflicts since this would needlessly complicate the computation of the correct local index (as well as needlessly complicate storing the shuffled keys in Step 7.0). As a result, runs of length *nWork*=[2,4,6,8] will experience [2-way, 4-way, 2-way, and 8-way] bank conflicts respectively and will add [1, 3, 1, 7] extra cycles on each shared memory access when loading the run. My current non-solution is just to live with any resulting bank conflicts.

However, as discussed in section 6.7.2, for runs of length [2,4, or 8], CUDA's low-level SASS compiler can optimize a sequence of two or four aligned sequential 32-bit load instructions into a single Vector2 (64-bit) or Vector4 (128-bit) load instruction. If this optimization is applied, then the number of load instructions and bank conflicts are both reduced by a factor of 2× or 4× since instructions that do not get executed do not cause bank conflicts.

Range Checks Issue: Since this `MapKeys` step stores sorted keys to the output array, four versions of this method are required to support the [‹FIRST?› ‹MIDDLE*› ‹LAST?›] range check pattern as part of my Row DASk:

> **`MapKeys_RC_BOTH`** does [*start*, *stop*] range checks, which is typically only needed for a input range smaller than a single fixed-size data block (*n* ‹ *DBS*).
>
> **`MapKeys_RC_START`** does [*start*, …) range checks, which is typically only needed if the *start* offset is not warp aligned to a warp boundary [0, 32, 64, …].
>
> **`MapKeys_RC_STOP`** does (…, *stop*] range checks which is only needed by the very last partially covered data block in the data set.
>
> **`MapKeys_RC_NONE`** does *NO* range checks, the vast majority of data blocks should end up calling the 'NONE' version, which is another reason to use my Row DASk.

STEP 9) `LoadValues` method:

The optional `LoadValues` method takes as input the *inValues* parameter and produces as output a short run of ‹*nWork*=[1-8]› sequential values stored in registers. This method is the same as the `LoadKeys` method from Step 1, except it works on values instead of keys.

STEP 10) `ShuffleValues` (Local Sort) method:

The optional `ShuffleValues` method distributes values from the current data block into sorted runs into a local array kept in shared memory. This method is almost the same as the

`ShuffleKeys` method from Step 7, except it works on values instead of keys and reuses the local starts already computed in Step 6.

STEP 11) `MapValues` (Global Sort) method:

The optional `MapValues` method distributes values into sorted runs in the output array kept in global memory. This method is almost the same as the `MapKeys` method from Step 8, except it works on values instead of keys, and it also reuses the global output offsets already computed in Step 8.

STEP 12) `UpdateRowStarts` method:

The `UpdateRowStarts` method accumulates d = 16 current *block-sums* [0-15] into the current *row-starts* [0-15] using sixteen active threads. After distributing all keys (and values) within the current fixed-size data block into their respective sorted runs in global memory, this method moves the current row-starts (global run positions) just past the distributed local data runs to get ready for sorting the next data block along the current data row assigned to this thread block.

9.4 Experiment Results

For my GPU Radix Sort, I focus in this section on experiments exploring the trade-offs for ILP and TLP to find the best performance results. All tests were performed on a GTX 580 (Fermi) and a GTX Titan (Kepler) using the host environment in Table 9.9.

CPU Hardware: CPU = i7-4770K@3.50 GHz, RAM=12 GB
GPU Hardware: *GTX 580* (16 SMs, 512 SPs, 1.5 GB RAM, 192.4 GB/s peak throughput) *GTX Titan* (14 SMXs, 2,688 SPs, 6.0 GB RAM, 288.4 GB/s peak throughput)
Software: GPU API = CUDA 5.5, C++, IDE = VS 2010, OS = Windows 7, SP1, Pointers = 64-bit
Data: Input size, $n = [2^{10} – 2^{27}]$, in increasing powers of two
Table 9.9: *Radix Sort Experiment Environment*

For these experiments, I measured performance using data throughput across the entire GPU radix sort. In other words, how many ‹*key*, *value*› pairs can my algorithm sort per second. Data throughput was measured in mega-pairs per second (Mpairs/s) by counting the number of pairs to be sorted (*n*, in millions) and dividing by the average run time for the entire radix sort to complete in seconds. My experiments found the best data throughput by varying the TLP and ILP parameters (similar to Section 6.8.1).

9.4.1 Data Throughput

To find the optimal data throughput, I varied both TLP and ILP using two template parameters, ‹*nWarps*›, ‹*nWork*›, respectively. For example, my *baseline* pair ‹*nWarps*=1, *nWork*=1› uses a single thread warp per thread block and sorts a single ‹key, value› pair per thread. I measured data throughput for increasing *n* (as powers of two) for both *nWarps* and *nWork* in the range {1..8} resulting in 64 (=8*8) possibilities. Tables 9.10 And 9.11 show the maximum data throughput for all 64 cases for the GTX 580 and the GTX Titan, respectively.

GTX 580		***nWork* = [1-8]**							
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
***nWarps* = [1-8]**	**1**	**119.07**	249.06	274.72	371.26	401.65	407.98	**475.46**	452.24
	2	207.95	419.51	451.64	596.04	597.75	648.21	661.97	**717.63**
	3	**260.80**	499.40	475.27	638.06	618.00	597.88	615.29	675.42
	4	253.20	489.02	499.11	564.30	574.42	618.15	639.34	666.62
	5	164.75	475.98	450.68	526.99	513.80	547.83	566.93	623.60
	6	237.06	427.36	430.48	569.12	545.91	489.82	507.17	560.36
	7	217.85	440.49	372.64	494.13	478.96	516.81	527.33	585.38
	8	186.05	373.60	388.09	509.72	488.57	521.47	539.36	588.19

Table 9.10: Maximum Data throughput on the GTX 580 for each of 64 experiments. Data throughput is measured in millions of pairs per second (Mpairs/sec).

GTX Titan		*nWork* = [1-8]							
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
nWarps = [1-8]	**1**	**186.28**	318.43	357.07	444.94	427.02	452.97	475.46	**489.76**
	2	311.28	521.35	599.50	690.68	672.38	707.42	727.32	742.07
	3	361.29	435.26	468.89	774.77	742.07	768.04	774.61	799.95
	4	**378.04**	452.31	690.67	795.61	763.38	811.04	816.93	**835.34**
	5	375.99	451.45	468.74	750.61	735.52	738.38	749.25	766.26
	6	367.21	432.03	480.62	776.51	732.82	730.98	743.97	767.88
	7	361.19	413.66	622.15	727.36	707.05	710.47	712.71	744.02
	8	357.29	587.34	595.09	702.68	796.31	755.37	763.20	795.74

Table 9.11: Maximum Data throughput on the GTX Titan for each of 64 experiments. Data throughput is measured in millions of pairs per second (Mpairs/sec).

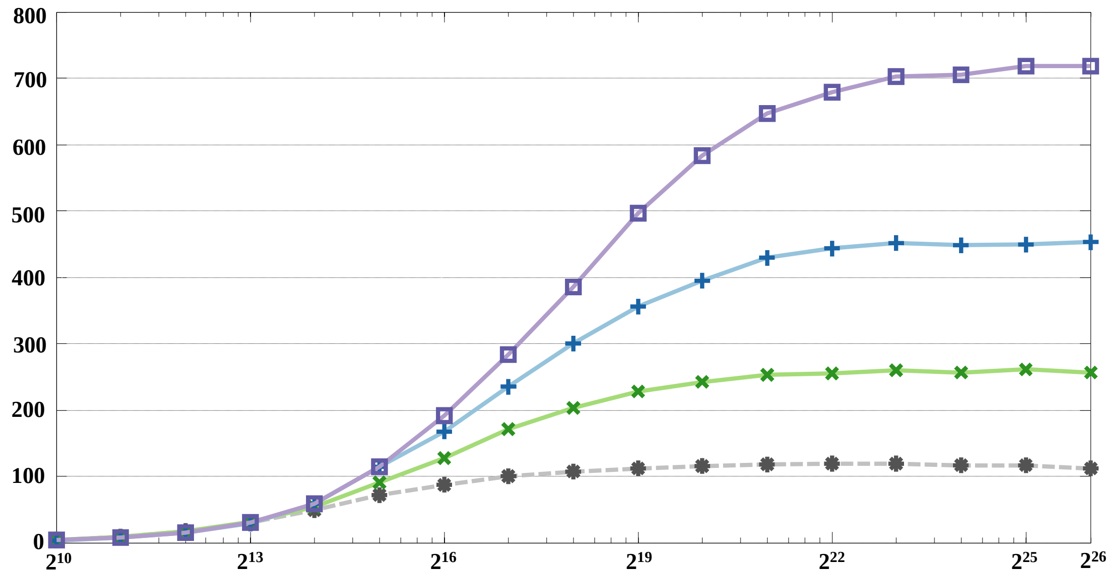
For both tables, each of the 64 ‹*nWork*, *nWarps*› pairs represents the maximum data throughput observed from a set of performance measurements that varies the input size of *n* = {2^8-2^{27}} in increasing powers of two. The first entry ‹1,1› represents baseline performance. The first row captures ILP performance by varying *nWork* in the range {1-8} and leaving *nWarps* fixed at {1}. The first column captures TLP performance by varying *nWarps* in the range {1-8} and leaving *nWork* fixed at {1}. The rest of the pairs represent the combined performance effect of both ILP and TLP in varying amounts.

Comparing the first columns and first rows, which capture TLP and ILP experiments, it is clear that ILP has a bigger impact than TLP. This makes sense, since most of the actual work is done in the `GPU_DistributeKeys` kernel and most of the work in that kernel occurs in locally counting and scanning keys (to prepare for a shuffle and map on each data block). Recall that scanning occurs hierarchically across four stages (S1:SS‹4›, S2:SS‹4›, S3:SS‹*nWarps*›, and S4:WS‹8›). Only in the first stage (S1) are all threads in the thread block participating. In stage 2 (S2), only four threads per thread warp are active, and in the last two stages (S3 and S4) only eight threads per thread block are active. So, the impact of TLP is lessened since only a few threads are doing useful work in these inner stages, while the rest of the threads are idle. On the other hand, ILP techniques can be applied across all twelve steps of the `BlockDistribute` method.

Best Throughput Experiments: I ran experiments for all 64 combinations of ‹*nWarps*, *nWork*›, each in the range {1–8}, to find the best overall data throughputs for Radix Sort were 717.63 for ‹2, 8› on the GTX 580 and 835.34 for ‹3, 8› on the GTX Titan. As discussed in Chapter 6, predicting winners on the GPU is difficult because of various constraints on occupancy caused by register and shared memory usage; based on my ILP and TLP test results, I would have predicted ‹3, 7› and ‹4, 8›, respectively. Figure 9.19 plots the data throughput results for four representative pairs for both the GTX 580 and GTX Titan.

GTX 580 Full RadixSort

Data Throughput (*Mpairs*/*sec*) vs. Input Size (*n*).



GTX Titan Full RadixSort

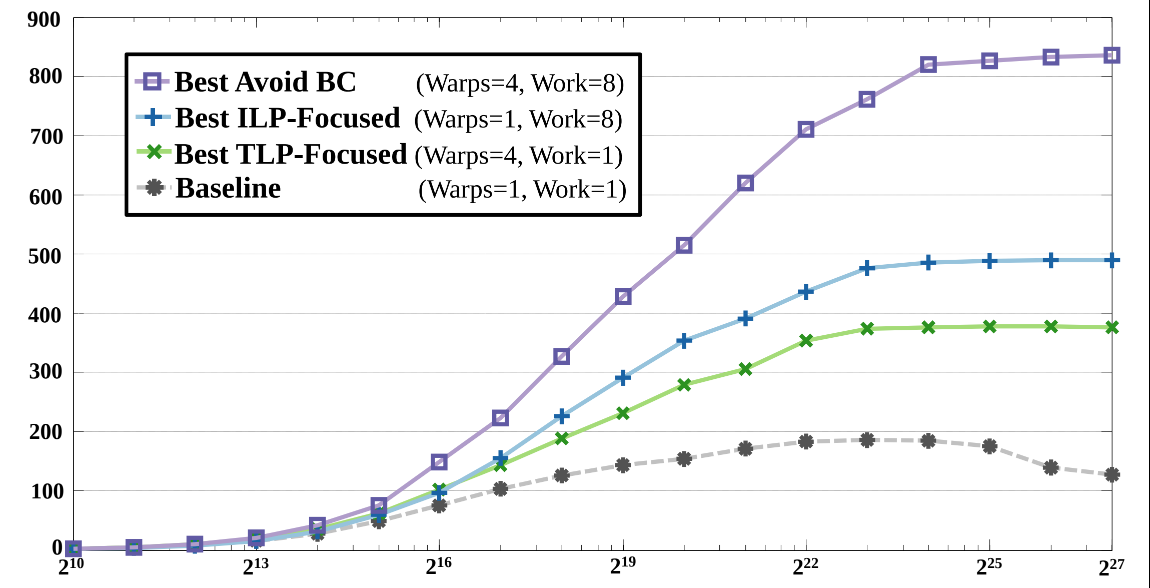


Figure 9.19: Full Radix Sort data throughput results (*y*-axis: millions of sorted pairs per second (Mpairs/s)) as a function of input size (*x*-axis; log-scale) on the GTX 580 {Fermi architecture} (top panel) and GTX Titan {Kepler architecture} (bottom panel) GPUs.

In both graphs, I chose the four pairs as the baseline: best ILP performance, best TLP performance, and best overall performance.

9.4.2 Total Cycles

In this section, I gather Total Cycle (*TC*) results from the NVidia Compute Visual Profiler (on CUDA 5.5) on the GTX Titan for $n = 2^{27}$ for both the `GPU_CountKeys` and the `GPU_DistributeKeys` kernels. The results are summarized in Tables 9.12 and 9.13 below.

Type Pair	Issued *II*	Issued *IPC*	T. Cycles *(II/IPC)*	*II/IPC Ratios*	*Grid*	*WI Eff. (None)*	*WI Eff (≥ 1)*	*Occupancy*
Baseline‹1,1›	83M	0.59	140M	1.00	224	90.61%	9.39%	13/64
TLP‹4,1›	91M	1.80	51M	2.76	168	69.84%	30.16%	40/64
ILP‹1,8›	63M	0.55	114M	1.24	224	92.80%	7.20%	8/64
Best‹4,8›	65M	1.38	47M	2.98	112	76.86%	23.14%	32/64

Table 9.12: Total Cycles (TC) performance from instructions issued (II) and average instructions retired per cycle (IPC) for the `GPU_CountKeys` kernel.

Type Pair	Issued *II*	Issued *IPC*	T. Cycles *(II/IPC)*	*II/IPC Ratios*	*Thput* (KV MP/s)	*Coalesced I/O efficiency*	*Bank Conflicts*	*Occupancy*
Baseline‹1,1›	1,189M	0.76	1,565M	1.00	186.28	6.75%	104M	16/64
TLP‹4,1›	766M	1.62	473M	3.31	378.04	18.15%	25M	48/64
ILP‹1,8›	399M	1.34	298M	5.25	489.76	26.81%	29M	16/64
Best‹4,8›	396M	1.71	231M	6.77	835.34	41.49%	38M	32/64

Table 9.13: Total Cycles (TC) performance from instructions issued (II) and average instructions retired per cycle (IPC) for the `GPU_DistributeKeys` kernel.

These two tables (9.12 and 9.13) measure the total cycles (TC) to complete a single Counting Sort pass within my Radix Sort for both the `GPU_CountKeys` and `GPU_DistributeKeys` kernels[12]. Recall that the total cycles can be computed from instructions issued (II) and average instructions retired per cycle (IPC) as TC = II/IPC (as discussed in Sections 3.1 and 6.8.2). Some notes about these tables are summarized below.

• The instructions issued (II), total cycles (TC), and bank conflict columns are measured in millions of instructions, cycles, or bank conflicts, respectively.

• The throughput column is measured in millions of sorted ‹*key*,*value*› pairs per second (MP/s). Note: These throughput results are for the entire radix sort (8 passes of counting sort on all three kernels) and thus shouldn't be compared directly to the total cycle performance results for each individual kernel.

• The II/IPC ratios column shows the relative speed-up of the other performance pairs (TLP, ILP, and Best) versus the baseline pair.

• The Grid column shows how many thread blocks are scheduled per kernel, this value is the same for both kernels. NOTE: These values are deliberately divisible by 14, which is the number of physical SMX cores on a GTX Titan.

• The Warp Issue Efficiency columns (WI Eff.) shows how often there were no active thread warps that could be scheduled onto the GPU SP cores (none) versus how often there were one or more (≥ 1) active thread warps that could be scheduled. These columns are not shown for the `K3_DistributeKeys` kernel to save space but the results are similar.

• The Occupancy columns show how many active warps were scheduled to run concurrently vs. the maximum of 64 thread warps that could be scheduled on a GTX Titan.

[12] My `K3_ScanRuns` kernel was also profiled but the results are not included as this kernel takes a trivial amount of time to complete compared to the other two kernels. The main interesting insight from the profile results for this kernel is that it is inefficient at taking advantage of GPU parallelism.

• The instructions retired per cycle (IPC) column has a maximum value of 5.0 on the GTX Titan. This value is impacted by both ILP and TLP.

• The bank conflicts column shows how many total bank conflicts are generated accessing shared memory. Note: Bank conflicts is not shown for the `GPU_CountKeys` kernel as they are measured in the hundreds not millions and do not significantly impact performance.

There are six significant insights that I see in the total cycles data.

• First, There appears to be a strong inverse correlation between total cycles and overall throughput. When total cycles is low, throughput is high (and vice versa). The comparison needs to be done carefully as throughput is aggregated across all 3 kernels.

• Second, in the `GPU_DistributeKeys` kernel, a high number of instruction replays caused by bank conflicts significantly increases total cycles and thus decreases throughput performance. Despite this, the Best‹4, 8› row still manages to have the best overall performance despite having 33% more bank conflicts than the ILP‹4, 1› row due to increased TLP parallelism.

• Third, the combination of both low instructions issued (II) and high instructions retired per cycle (IPC) results in the overall throughput winners. For instance in the `GPU_DistributeKeys` kernel, the TLP‹4, 1› row has great instructions retired per cycle (IPC = 1.62), but it still performs poorly because it issues so many instructions (II = 766 million). On the other hand, the ILP‹1, 8› row has a small number of instructions issued (II = 399 million), but it also performs poorly because it has the smaller instructions retired per cycle (IPC = 1.34). The Best‹4, 8› row has the best overall performance because it issues the fewest instructions (II= 396 million) and also retires the most instructions per cycle (IPC = 1.71).

• Fourth, Larger data blocks result in longer coherent runs on output which results in increased Coalesced I/O efficiency, fewer replays, and thus higher throughput. The data block size is not shown to save table space but for the four rows [baseline, TLP, ILP, and Best] is [32, 128, 256, and 1024] respectively. In the baseline case, each sorted data warp only successfully outputs about 6.75% of a data warp, which means about 14.8 stores are required on average to store each sorted data warp. Whereas in the Best case, each sorted successfully outputs about 41.50% of a data warp, which only requires about 2.4 stores on average per sorted data warp.

• Fifth, Both ILP and TLP help increase the number of active thread warps for the scheduler (GPU Hardware or CUDA software) to choose between to keep the SP cores busy doing useful work. However, even in the best case on the GPU_CountKeys kernel, we are only keeping the cores busy about 25% of the time, meaning that 75% of the time the scheduler must stall instead of issue a useful instruction to the SP cores. Similarly on the GPU_DistributeKeys kernel the Warp Issue Efficiency (One or more) percentages are is [16.12, 34.55, 24.97, and 32.46] respectively. So, even in the best case we are only keeping the cores busy ~35% of the time. This means there is an opportunity for better performance by rewriting code to do a better job avoiding stalls.

• Sixth and finally, The `GPU_DistributeKeys` kernel easily takes the most time. For example in the Best case, The `GPU_CountKeys` kernel costs about 47 million total cycles, while the GPU_DistributeKeys kernel costs 231 million cycles. Thus to improve overall Radix Sort performance, the primary focus should be on speeding up the `GPU_DistributeKeys` kernel.

The formula for total cycles (*TC* = *II*/*IPC*) tells me that there are two ways to attempt to improve performance by decreasing TC and thus improving throughput:

- Decrease instructions issued (II) by trying different algorithms or simplifying code
- Increase instructions retired, by increasing ILP via software pipelining and increasing TLP via higher occupancy, to consume as many instructions per cycle (IPC) as possible

To me, the main lesson is to keep parallel processing cores (SMs and SPs) and memory controllers as busy as possible. TLP keeps the processing cores busy by providing lots of active thread warps for the SM on-core scheduler to switch to when the currently executing warp stalls. ILP keeps the processing cores busy by providing lots of independent instructions for the scheduler to harvest to keep the instruction pipeline as full as possible. Fortunately, both approaches are orthogonal. Therefore, both techniques (TLP and ILP) can be used to hide pipeline and I/O stalls and keep the processing cores as busy as possible. The low percentages in the warp efficiency column tells me there is still lots of headroom for further improvement on my current performance results.

9.5 Conclusion

In summary, My Radix sort is a hybrid CPU/GPU sort. At the top level, the CPU iterates over multiple counting-sort passes (one per 4-digit radix) in LSD order to execute a full radix sort. Each counting-sort pass is implemented on the GPU using three kernels (`GPU_CountKeys`, `GPU_ScanRuns`, and `GPU_DistributeKeys`), which mirrors the same pattern as my Scan primitive (section 6.5). Both the `GPU_CountKeys` and `GPU_DistributeKeys` kernels are built using my ROW DASk.

The key idea is to partition keys into hierarchically sorted runs (at the thread-block level, at the thread-warp level, and at the individual thread level) with one sorted run per digit value. This partitioning allows each block, warp, and thread in the hierarchy to safely distribute their assigned ‹*key*, *value*› pairs into their final sorted positions without colliding with other blocks, warps, and threads concurrently distributing at the same time.

With the `GPU_CountKeys` kernel, each thread block marches along its assigned data row, data block by data block, counting all the keys along each row into a per-row count histogram containing (d = 16) row-counts. With the `GPU_ScanRuns` kernel, a single thread block scans all the row-counts into row-starts. This action not only partitions the output range [*start, stop*] into sixteen sorted runs (one per digit value) but also sub-partitions each sorted run into r smaller per-row runs (one per thread block). The resulting scanned row-starts allow each thread block to safely distribute its assigned data into its own individual block-runs. Finally, with the `GPU_DistributeKeys` kernel, each thread-block marches along its assigned data row, data block by data block, distributing ‹key, value› pairs into sorted runs using the complex twelve-step `BlockDistribute` method on each data block in turn. This `BlockDistribute` method counts keys, hierarchically scans counts into starts, and then distributes keys (and values) into sorted runs.

To make this all work, I had to deal with several issues that hindered performance: coalescence, bank conflicts, grid size, register pressure, and shared memory pressure

- **Coalescence:** To respect coalescence for better global memory throughput, I needed two different solutions: one for input and one for output. For input, I transferred data between global memory and registers using the warp by warp access pattern. I then converted data into a sequential access pattern using shared memory in two more steps. For output, I shuffled (locally sorted) the data in shared memory first to increase run coherence. This extra shuffle step reduced the number of store operations required to distribute the local sorted runs into their final sorted positions. However, the impact of this local sort, as measured by my “coalesced I/O efficiency” metric, depends on the average local run length depends on the data block size (DBS) and the number of digit values (d=2^4), with *runLength* = *DBS*/*d*. In general, I found that the longer the coherent run being output, the more efficient coalescence is and the fewer stores operations are required to distribute the run into global memory.

- **Bank Conflicts:** To avoid bank conflicts when accessing shared memory (which can cause serialized replays), I tried using odd length runs, utilizing the *“Pad & Rake”* technique on power of two length runs; aligning Vector4’s so that CUDA can decrease bank conflicts by a factor of 2× or 4×; and, finally, just living with a low number of serialized replays caused by bank conflicts.
- **Grid Size:** My Row DASk requires small grid sizes, which makes my solution sensitive to picking a good work load (as described in section 6.7.3), which evenly divides the thread blocks in the work load across the SM cores on each GPU card. Picking a good work load requires that the programmer understands constraints on occupancy and picks the initial grid size accordingly.
- **Register Pressure:** My current code multi-scans sixteen sets of counts into sixteen sets of starts. I compressed four 8-bit counters per 32-bit lane to help decrease this scan pressure somewhat. However, compressed data requires decompression to access. Decompression increases the number of instructions and registers in other ways. In addition, I used software pipelining in batches of [2,3,4 or 8] throughout my code to increase ILP performance. Both multi-scan and ILP techniques increase register pressure. My current code takes anywhere between 38 and 63 (+10) registers as *nWork* varies between [1-8], where the (+10) refers to additional spill registers

(using global memory to store extra registers which negatively impacts performance). This register pressure negatively impacts occupancy reducing the number of thread-blocks that can run concurrently.

- **Shared Memory Pressure:** I have written many different versions of Radix Sort seeking to improve performance. Almost all of my earlier solutions used 2-3× as much shared memory as my current solution. This caused my shared memory usage to constrain occupancy (more than register pressure). I made three main changes to decrease shared memory usage. First, I transitioned from a block-share pattern to a warp-share pattern. In the block-share pattern, I treated memory arrays as accessible by every thread in the block all the time. In the warp-share pattern, I divided memory into individual per-warp sub-buffers, where each warp can work on its assigned memory sub-buffer independently. This allows memory use to grow dynamically with the *nWarps* parameter as it varies between [1-8]. Second, I figured out how to overlap the S1 and S2 scan arrays during my scan which reduced the total size of my scan arrays by 25%. Third, and most importantly, I carefully reused the same memory for different purposes at different times in my algorithms.

By increasing TLP, increasing ILP, using efficient I/O access patterns, mitigating bank conflicts, and reducing memory space, I improved Radix Sort performance over my baseline performance. The best performing throughput for my Radix Sort primitive was up to 6.1× and 4.4× faster than the baselines on the GTX 580 and GTX Titan, respectively.

9.6 Future Directions

I estimate that the maximum peak rates[13] of sorting 32-bit ‹*key*, *values*› pairs on the GTX 580 and GTX Titan are 1205 million and 1802 million pairs per second, respectively. My GPU radix sort implementations achieved 59.5% and 46.3% of peak data throughput on the GTX 580 and GTX Titan, respectively.

These performance results are solid but fall well short of the maximum throughput rates possible on these architectures. In other words, my Radix Sort method still has limitations that decrease performance. My method has high register pressure, which constrains occupancy. It performs many compression/decompression operations to save registers but which increases the total number of instructions. Perhaps a different approach may lead to greater performance gains. I have four ideas for future investigation: specializing code, rewriting code to take full advantage of the Vector4 optimization, decreasing passes using a larger radix, and writing a specialized WarpScan using the PTX Shuffle command.

Specialize Code: My current code was written in a generalized way to support experiments on ‹*nWarps*, *nWork*› pairs, both in the range [1-8]. My experiments show that that the best performance for the GTX 580 and GTX Titan occurs on the pairs ‹2,8› and ‹4,8›, respectively. Rewriting hard-coded specific versions of code for these specific pairs may allow various short-cuts to be taken, which should result in lower register use and fewer instructions required, unlocking more peak performance.

Shared Memory access using Vector4: My code could be optimized to use Vector4 access into shared memory as much as possible. This replaces four sequential shared memory accesses (loads/stores) with a single Vector4 (128-bits) instruction instead and therefore reduces memory

[13] Given a peak I/O throughput of 192.4 and 288.4 GB/s for the GTX 580 and 288.4 respectively. A 32-bit key (and value) each take up 4 bytes each. 32-bit keys also requires a total of 8 sorting passes (8=32/4) for a 4-bit radix. Assuming 5 I/Os per pass to sort (1 read to count + 1 read/write per key to distribute + 1 read/write per value to distribute). Then I compute the maximum sorting rate for the GTX 580 as (192.4/(4 bytes * 8 passes * 5 I/Os per pass), which results in 1205 million pairs per second. A similar calculation for the GTX Titan (288.4/(4*8*5) results in 1802 million pairs per second.

access instructions and bank conflicts by a factor of 4×, wherever this optimization is applied.. To leverage this optimization, shared memory accesses must be aligned to Vector4 boundaries (128-bits). This optimization means that the "Pad and Rake" technique cannot be used to mitigate bank conflicts. **Note:** Based on my own painful experiences with poor I/O throughput, the Vector4 optimization should not be applied when accessing global memory.

Reduce the number of Counting Sort Passes: In order to sort 32-bit keys, my code currently needs to sort in eight passes (=32/4). CPU Radix Sort code typically uses 8-bit radices in order to fully sort data in only four passes (=32/8), greatly reducing the total number of I/Os required. (I wrote two completely different versions of GPU Radix Sort that used 8-bit radices to sort. Unfortunately, due to high pressure on registers and shared memory, both versions were compute-bound, taking 24-32 complex steps to distribute and my best performance was around 500 million pairs per second).

Use PTX "Shuffle" Command: Recent GPU architectures (CC ≥ 3.5) like the GTX Titan (CC ≥ 3.5) support the new "SHUFFLE (SHFL)" PTX command, which allows warp-scans across all threads of a thread-warp. My `ScanWarps` step could be rewritten to scan four data warps at once taking advantage of the SHUFFLE command for potentially better performance.

9.7 Lessons Learned

Future directions includes ideas about how to improve radix sort performance. However, there are many practical lessons I learned from my experiments on radix sort using my Row DASk. This section summarizes many of these lessons.

`RadixSort` Lessons:

- To support sequential processing
  - Use my Row DASk to support sequential processing across thread blocks, which is required by the "Distribute" step in Counting Sort.
  - Group *n* keys into *m* fixed-size data blocks, $m = \lceil n/DBS \rceil$

  - Partition the *m* data blocks across *r* rows, where *r* is picked by the programmer based on the expected work load, resulting in *c* columns, *c* = $\lceil m/r \rceil$.
  - Partition fixed-size data blocks into short sequential per-thread runs to support sequential processing within thread blocks (across warps, across threads). **Note:** Hierarchal operations within each "distributed" data block should avoid reordering count, scan, and distribute operations within data blocks, and thread runs as the commutative property cannot be safely assumed.
- To increase performance via ILP
  - Work on multiple [2-8] work items per thread. This is similar to loop unrolling on CPUs. **Note:** This approach increases register pressure, since *k* work-items require *k*× as many registers.
  - Use software pipelining to regroup and reorder instructions from multiple work items. **Note:** Watch out for code that cannot be software pipelined safely (like incrementing counters within a histogram by multiple work items).
- To increase performance via SIMD
  - Launch thread-blocks that are a multiple of the *WarpSize* [32, 64, …, 1024], since the GPU SIMD hardware is designed to efficiently support warp-level processing.
  - Launch [2-4] thread warps per thread block, since the hardware supports multi-issue dispatch.
- To increase performance via TLP
  - Launch as many thread blocks as possible.
  - Pick the number of thread blocks, rows (*r*), to be a multiple of the work load (see section 6.7.3) to evenly divide thread blocks across SM cores.
  - Maximize occupancy given current constraints on occupancy (hardware limits, register pressure, and shared memory pressure).
  - Decrease occupancy constraints (reduce register & shared memory pressure)
    - Rewrite code to decrease registers.
    - Software pipeline in batches of [2-4] work items to decrease registers.
    - Redesign memory layouts to decrease shared memory use
      - Divide memory into per-warp and per-block buffers.
    - Reuse memory (same buffer for many purposes).
      - Same per-warp memory buffer for count, scan, and distribute.
      - Reuse the S1 scan array for both the S1 & S2 scans.
    - Compress data in registers and shared memory. **Note:** This increases code complexity in order to compress/decompress required values.
    - In my radix sort, register pressure is the main constraint on occupancy.
- To increase I/O throughput via Coalescence
  - Load input data using my warp-by-warp BASk (see section 5.1.1.) in warp-aligned runs of 32 data elements for 100% coalesced I/O efficiency.
  - Convert between a warp-by-warp view of global memory and a sequential view of data required for a stable counting sort using a three-step conversion method (see section 6.7.1).

  - Store output data in long runs of coherent data. The longer the data run, the higher the "coalesced I/O efficiency" on output.
- Mitigate bank conflicts in shared memory to avoid lost cycles due to serialized replays
  - Use odd-numbered run lengths [1,3,5,7].
  - Use the "Pad & Rake" technique for run lengths that are powers of 2 [2,4,8].
  - Take advantage of the Vector4 optimization to reduce bank conflicts by 4×.
  - Live with the resulting bank conflicts (run length = [6]).
    Note: Some bank conflicts are unavoidable (unpredictable access patterns)
    Note: Whack-a-mole (when multi-step methods have different views of the same memory in different steps then mitigating bank conflicts in one step may cause them to pop up in another step.)
- To increase Scan performance
  - Use a hierarchical multi-level nested multi-scan to scan an entire data block (For instance: S1:SS‹4›, S2:SS‹8›, S3:SS‹*nWarps*›, S4:WS‹16›). NOTE: this results in idle threads and warps at higher levels of the hierarchy.
  - Use the scan-then-fan pattern (see section 6.4), and preserve short runs in registers between stages to decrease the total summations and shared memory accesses required.
  - Serially scan short runs in registers using Sklansky's method (Section 6.6.2).
  - Cooperatively scan across threads using the `WarpScan` method (Section 6.6.3).
  - Scan inclusive then store exclusive. Exclusive scans can be built from inclusive scans by reaching back one column.
  - Compress data, so multiple counts can be scanned using bit-level parallelism. **Note:** Overflow can result requiring code that upscales data before summing.
- To increase performance via the Total Cycles ($TC = II/IPC$) metric
  - Minimize the number of instructions issued (II).
    - Eliminate unnecessary instructions
    - Rewrite code to simplify complex calculations
  - Maximize the average instructions retired per cycle (IPC).
    - Software pipeline multiple work items to increase ILP.
    - Rewrite code to decrease instruction dependencies and hardware stalls.
    - Launch many concurrent thread blocks (and warps) to increase TLP.
    - Mitigate constraints on occupancy to increase TLP.
- Miscellaneous Lessons
  - To coordinate data communication across thread warps, use barrier synchronization (`syncthreads`). **Note:** This is the only mutual exclusion primitive needed for correct parallel behavior in my GPU radix sort.
  - For better performance, as part of kernel setup, pre-compute some view indices or pointers (warp-view and sequential-view) used in the `BlockCount` and `BlockDistribute` methods. This amortizes the higher cost of "Raking."
    Note: storing these pre-computed indices or pointers increases register pressure.
  - Since the Row DASk requires four sets of very similar code to properly support amortized range checking, watch out for copy and paste errors in the various

`BlockDistribute` methods. If fixing a bug in one version, remember to check and fix the other versions as well.

- For GPU Counting Sort, the `GPU_CountKeys` and `GPU_DistributeKeys` kernels both must use the same data rows for correct scan results. My solution is to have both kernels use the same Row DASk, the same CTA layout, and the same DBS.
- For GPU Counting Sort: For fewer global I/Os (3× instead of 4×) at the kernel level, follow the reduce-then-scan pattern, and avoid the scan-then-fan pattern (see section 6.4). Using the Row DASk on both the `GPU_CountKeys` and `GPU_DistributeKeys` kernels naturally supports this pattern.
- For CPU radix sort, swap between input and output pointers (ping-pong) to avoid unnecessary memory copies after each Counting Sort pass.

10.0 Conclusion

Programmers have many data structures and algorithms to choose from when they solve a real-world problem. Experienced programmers select their algorithm by balancing between competing demands and constraints of the problem. Great programmers also understand the underlying hardware and select the algorithm and implementation to take advantage of beneficial hardware features while avoiding hardware constraints. I have seen great programmers achieve ten-fold increases in performance over a straight-forward implementations of an algorithm.

The many competing constraints make GPU parallel programming hard, even for the most experienced programmers. They typically begin with a good single-threaded serial algorithm and convert it into a massively multi-threaded parallel algorithm that is both correct and robust. GPU programming is made more difficult, since programmers must decide how to map millions of data elements onto tens of thousands of threads, represented by a complex two-level parallel hierarchy called a cooperative thread array (CTA). Increasing the burden yet further, programmers need to understand the complex GPU memory model to maximize data and I/O throughput. Picking the right memory access pattern to enable peak throughput can be difficult, but, if done right, it can result in a massive parallel performance increase.

To increase parallel performance, programmers must understand the GPU's model of data-level parallelism (SIMD), pipelining and instruction-level parallelism (ILP), and multi-threading and thread-level parallelism (TLP). For high-performance GPU implementations, programmers need to understand the resource trade-offs imposed by GPU architecture--with its multi-level memory hierarchy, the two-level cooperative thread array (CTA) parallel hierarchy, and other GPU hardware issues such as coalescence, bank conflicts, and occupancy constraints. GPU programmers must also be able to conduct experiments to find the best performing set of parameters on a particular GPU device. In other words, a beginning GPU programmer must master a daunting set of skills to become great.

To help facilitate this mastery, I have developed memory layout and access frameworks that provide GPU programmers with a starting point for developing their own specific parallel implementations to solve general problems on the GPU. I call these frameworks data access skeletons (DASks) and block access skeletons (BASks). These frameworks provide a strong starting points for converting single-threaded serial algorithms into high-performance parallel implementations on GPU machines.

10.1: DASks and BASks

As discussed in Section 2.2.1 and Section 3.3.1, modern GPU architectures use a 2-level parallel hierarchy: a set of MIMD SM cores, each containing a set of SIMD SP cores. Parallel warp-threading also has a 2-level hierarchy: thread blocks within a grid, threads within each thread block. Memory has a configurable hierarchy as well. In addition to global memory, programmers can choose how to allocate shared memory and registers, subject to hardware constraints. Threads and memory interact in complex ways in these hierarchies. To reduce the complexity that programmers must consider, I produced three data access skeletons (DASks) and two block access skeletons (BASks), as discussed in Chapter 5. These skeletal frameworks are higher level abstractions for memory access patterns that map well onto the GPU. The DASks partition data arrays (input and output) into fixed-size data blocks then load-balance these data blocks across thread blocks using different layout patterns. The BASks are generalized memory access patterns for efficient coalesced I/O throughput at the thread-block level.

The skeletal frameworks are next parameterized to support experiments on the best combinations of instruction and thread-level parallelism as well as to create the best balance between shared memory and registers. This dissertation has demonstrated such experiments in several case studies. As shown in my case studies, each DASk provides the general skeleton for parallel processing over data blocks, and I, as the GPU programmer, then provide a “body” method to solve a specific problem (such as Copy, Reduce, Radix Sort) on a fixed-size data block. GPU programmers can then take my existing “body” sections and replace them with code to solve their own problems with high confidence since the underlying framework code forming the skeletons has already been tested, works correctly, and achieves high throughput.

I report many implementation details for the case studies of the previous chapters. Therefore, in this conclusion, I highlight two broad insights that might have gotten lost among those details: partitioning and fixed-size data blocks.

Partitioning:

As discussed in Chapter 5, partitioning[1] is where my DASks group *n* data elements into *m* fixed size data blocks. These *m* data blocks are then load-balanced across *p* thread blocks using different memory access patterns. My DASks provide the “skeleton” for accessing data blocks according to a 1D or 2D layout. GPU programmers then provide the “body” methods for processing individual data elements within each data block. Unfortunately, correct use of partitioning requires that all data is known a priori before executing the GPU algorithm and that the input set remains static during the course of executing the GPU algorithm. Consequently, algorithms on unknown, streaming, or dynamically varying data sets will require a different approach. There are five important benefits to partitioning: parallel processing, data coherence, sequential ordering, data independence, and deterministic algorithms.

Parallel Processing: By design, my DASks are built to efficiently support the 2-level cooperative thread array (CTA) hierarchy (thread blocks within a grid and threads within a thread block). This enables programmers to take advantage of the tens of thousands of threads on each GPU for massive parallel performance.

[1] What I describe as “partitioning” is also known as “data decomposition” or “geometric decomposition”.

Data Coherence: By grouping data into fixed size data blocks, data elements are kept localized within each data block, resulting in high coherence. Loading warp-aligned input elements within each data block via the Block-by-Block or Warp-by-Warp BASk can take full advantage of coalescence for 100% coalesced I/O efficiency and peak I/O throughput. When possible (as in Scan from Chapter 6), outputting coherent data blocks of results allows peak I/O throughput as well. When not possible (as in Radix Sort from Chapter 9), outputting longer runs of coherent results increases coalesced I/O efficiency and allows for a solid fraction of peak I/O throughput.

Sequential Ordering: For algorithms (such as Scan and Radix Sort) that require data to be processed in a sequential order for correct behavior, my Row DASk supports sequential ordering at the data block level. Within each data block, GPU programmers can load and convert *k* data warps per thread warp into per-thread sequential runs of *k* elements using my 3-step conversion process, as described in section 6.7.1.

Data Independence: With one exception[2], all my case study algorithms are data independent. In other words, for a given fixed input size (*n*), changing the values of data within the input data set has no noticeable impact on performance.

Deterministic Algorithms: All of the GPU primitives implemented in my case studies are deterministic algorithms. They are lock-free and avoid all atomic operations. For correct parallel coordination, the only mutual exclusion primitive actually used is barrier synchronization[3] across all threads within a thread block (as discussed in Section 2.2.3). As a result, there are no parallel communication slow-downs[4] due to waiting on locks or atomics, which can serialize parallel performance. Also, there are no concurrency side effects, such as incorrect results, deadlocks, or

[2] The one exception is my nearest neighbor *k*d-tree case study from Chapter 7, this primitive unfortunately is data dependent. The location of query data points within the search tree causes different amounts of branch divergence within each thread warp, which causes overall performance to fluctuate as data elements vary within the data set.

[3] My barrier synchronization uses the CUDA `syncthreads` method.

[4] Barriers are not a complete panacea, as barriers still can result in idle cycles by finished thread warps which must wait on other thread warps to complete.

starvation, which often require complex code to carefully resolve. This is a direct result of my decision to base my DASks on partitioning data across the 2-level CTA hierarchy. Once all threads have identified the unique locations required to load its inputs and store its outputs, each thread can then proceed at full speed without needing to coordinate with the other tens of thousands of parallel threads each processing their own data at the same time.

Fixed-Size Data Blocks:

Fixed-size data blocks are a fundamental unit of work for efficient GPU data parallel programming. By construction both my BASks and DASks first organize data into short fixed-size runs, each processed by a single thread. These runs are then organized into fixed-size data blocks, each processed in parallel by a fixed-size thread block of multiple thread warps. This organization allows all levels of the GPU parallel hierarchy to avoid idle time: The SP cores process a warp of threads at a time, the SM warp schedulers have many warps to switch between on stalls for I/O or memory access, and the individual threads have enough computation between memory accesses to make progress.

The block sizes are determined by three parameters: *WarpSize*, *nWarps*, and *nWork*. At the bottom level of the GPU parallel hierarchy, the SP cores process a warp of *WarpSize* (=32) threads to take full advantage of SIMD. My DASks enable programmers to boost TLP on the SMs (and hide memory latency) by choosing the number of warps in a small range, ‹*nWarps*=[1-8]› to get the fixed *thread block size, TBS* = *nWarps***WarpSize*. My DASks also enable programmers to increase ILP by choosing the amount of work per thread in a small range, ‹*nWork*=[1-8]›, to get the fixed *data block size, DBS* = *nWork***TBS*. Consequently, the data block size varies as a function of the ‹*nWarps*, *nWork*› template parameters chosen at compile time. Generic programming techniques (C++ templates, inlining, etc.) allow the resulting compiled code to adapt to the chosen template pair ‹*nWarps*, *nWork*›. This ‹*nWarps*, *nWork*› template pair[5] enables experiments on parallel performance (ILP and TLP) to find the best performing DBS for a given problem, as shown in the case studies for Copy in Chapter 5, Reduce/Scan in Chapter 6, and Radix Sort in Chapter 9.

10.1.1 Block Access Skeletons (BASks):

To improve I/O performance within each fixed-size data block, I produced the *Block-By-Block* and *Warp-By-Warp block-access skeletons* (BASks) (shown in Figure 10.1) to transfer data between global memory and registers.

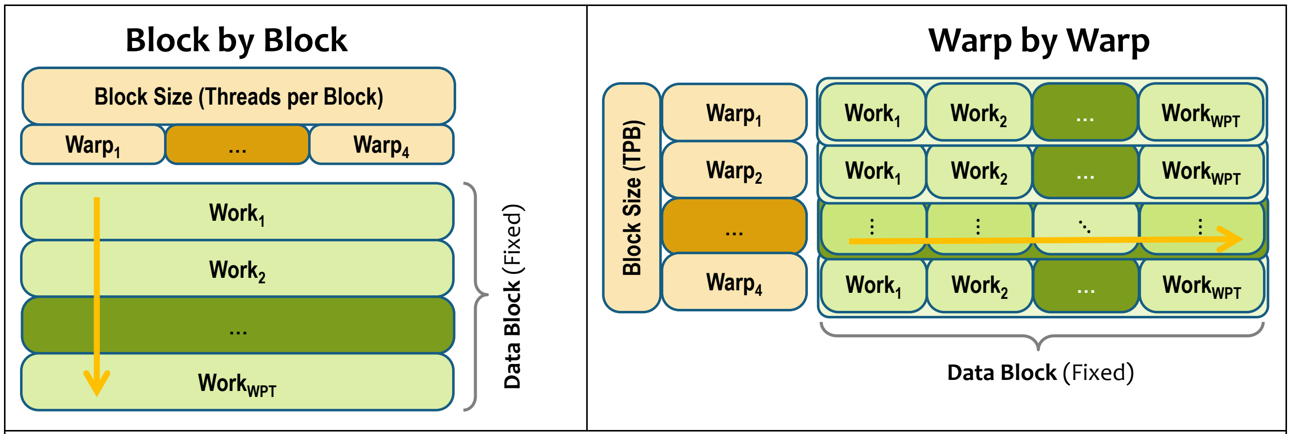


Figure 10.1: The *Block by Block* and *Warp by Warp* BASk layouts respectively.

As discussed in Section 5.1.1 and shown in Figure 10.1 (left panel), the *Block-By-Block* access pattern strides through the data block cooperatively, using all threads in the thread block. In other words, each thread within the block accesses its own unique data element within the current group of TBS elements and then strides (*stride* = *TBS*) to the next group of TBS data elements within the data block to access. As also discussed in Section 5.1.1 and shown in Figure 10.1 (right panel), the *Warp-by-Warp* access pattern assigns each thread warp its own unique

[5] *WarpSize* is also specified as a C++ template parameter to allow for future changes but for all current generations of modern GPU architectures I have worked on (Tesla, Fermi, Kepler, Maxwell) it has remained fixed at 32 threads per warp.

chunk of data within the data block, and then each thread warp strides through its assigned data chunk one data warp at a time (*stride* = *WarpSize* = 32).

Although the *Block-By-Block* pattern is simple, easy to code, and easy to reason about, I prefer the more complex *Warp-by-Warp* for three main reasons:

- **Coherence:** The underlying access pattern is slightly more coherent (localized), which can result in a modest increase in I/O throughput (1-2% faster) despite the extra instructions required for setup and indexing.
- **Warp independence:** Each individual thread warp can proceed to work on its assigned chunk of data independent of any other warp in the thread block.
- **Less synchronization:** The Warp-by-Warp pattern avoids extra Barrier instructions that would be required for correct parallel results in the Block-by-Block pattern.

The main take-away for programmers is to use fixed-size data blocks that are multiples of the desired work and warps for a given problem and to use a Warp-by-Warp access pattern into global memory when processing each data block.

10.1.2 Data-Access Skeletons (DASks):

To take advantage of the top layer of the CTA parallel hierarchy, I produced three data access skeletons (DASks): *Block*, *Column*, and *Row* DASks. All three of these DASks partition by first grouping *n* data elements into *m* fixed-size data blocks and then load-balancing the *m* data blocks across *p* thread blocks. As discussed in Section 6.7.3, for better performance, programmers should be careful to choose the initial grid size so that it balances work loads of concurrently running thread blocks evenly across the all the SM (or SMX) cores on a GPU card.

Block DASk:

As discussed in Section 5.1, my *Block* DASk (shown in Figure 10.2) follows a simple 1D layout pattern, where *m* data blocks are mapped onto *m* thread blocks.

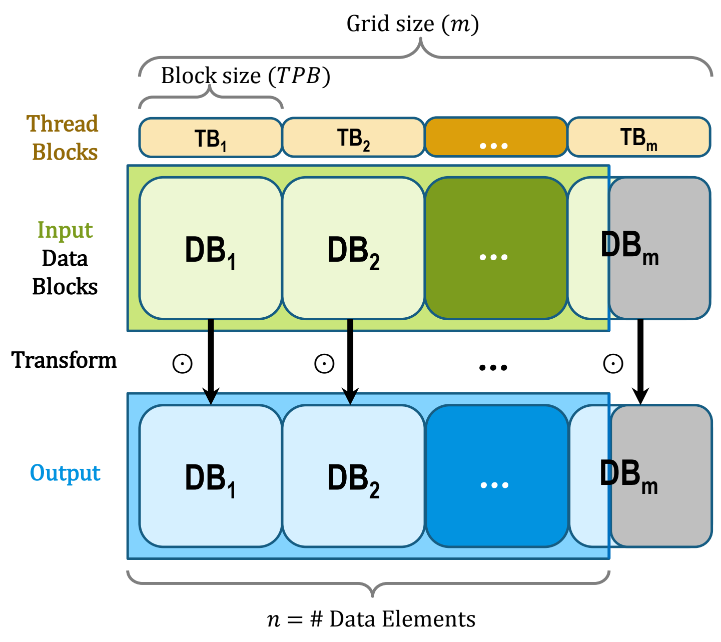


Figure 10.2: The *Block* DASk layout.

My *Block* DASk (shown in Figure 10.2) enhances the simple Copy I/O kernel (from Chapter 4) with C++ template parameters, multiple work items per thread, amortized range checking, amortized pointer indexing, and manual loop unrolling. The Block DASk follows a 1D layout pattern on input (and output) arrays by grouping *n* data elements into *m* fixed size data blocks and then assigning one thread block per data block to process data. The last fixed-size data block in the input array may only be partially full, requiring range checks to avoid out-of-range memory accesses. Using multiple work items per thread allows programmers to amortize the costs of range-checks and pointer setup across all ‹*nWork*=[1-16]› work items in each data block, which decreases the total number of range checks required by (1/*nWork*).

Experiments reported in Section 5.1.6 suggest that for simple kernels, like Copy, TLP has a much more pronounced impact on increasing throughput than ILP. TLP experiments for thread block sizes (TBS) in the range [32-1024] reveal the significant effect of occupancy constraints on the number of concurrently running thread blocks. The best performing thread blocks were found at TBS=[128, 256, 512, and 1024] with TBS=128 (four warps). These had the best overall performance. Experiments on ILP reveal that software pipelining has greater potential to increase throughput than simple loop unrolling, which is not unexpected. My *Block* DASk is recommended for simple transformation kernels (such as Fill, Copy, Gather, and Scatter) on linear arrays, since the setup costs are low.

Column DASk:

My *Column* DASk (shown in Figure 10.3) follows a 2D layout where *m* thread blocks are load balanced across *c* fixed columns into *r* rows.

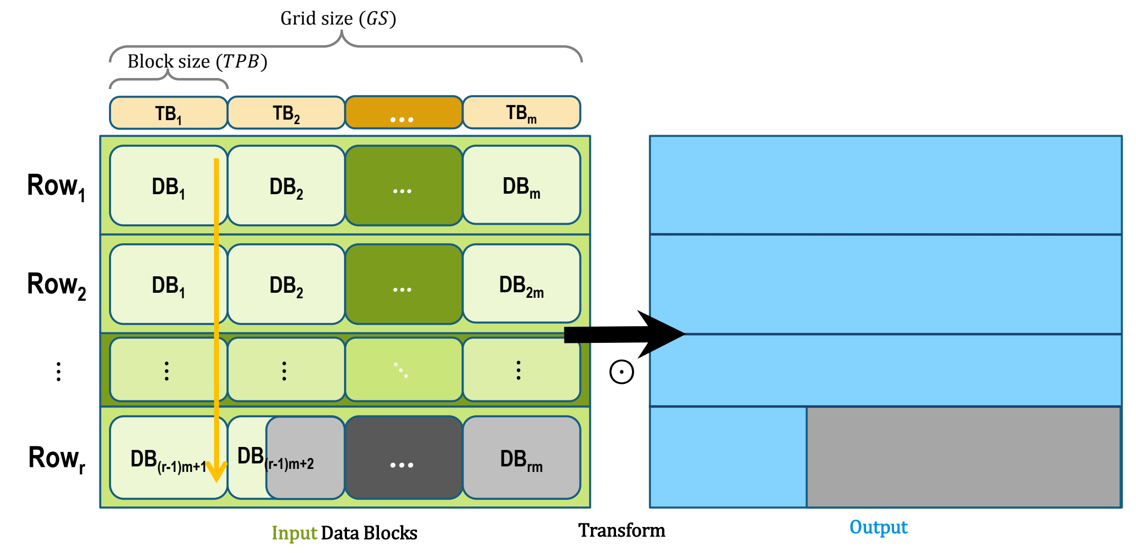


Figure 10.3: The *Column* DASk layout.

As discussed in Section 5.3, the *Column* DASk uses a two dimensional data access pattern. The input data (*n*) is grouped into *m* fixed-size data blocks (DBS) with the data blocks being laid out on a 2D grid, with *m* = $\lceil n/DBS \rceil$. The number of columns (*C* = *grid.width*) is fixed and chosen ahead of time by the GPU programmer. The number of rows (*R* = *data blocks per column*) varies with *m* (and thus *n*) in order to fully cover all input data, with *R* = $\lceil m/C \rceil$. The last row may be partially full, requiring range checks to prevent out-of-range data access. The full rows require no range checks. The Column DASk maps one thread block onto each data column, with each thread block being responsible for processing all data blocks along its assigned data column. All thread blocks cooperatively stride through the entire data set, one row at a time. The *Column* DASk is conceptually similar to the Block-by-Block DASk but works across the entire data set, rather than just on one fixed-size data block.

The column DASk has three main benefits and five main limitations.

Benefits:

- It supports proper range checking of all data [*start*, *stop*] while only requiring range checks on the last row of data; the cost of doing the range check on the last row of data is amortized across the other full data rows within the column.
- It supports an efficient access pattern into memory using a 2D grid layout.
- Is works well when processing data where the underlying data order doesn't matter (example: Reduce, Histogram).

Limitations:

- GPU programmers need to write two slightly different versions of the same code to handle the three different range check cases {*in-range*, *out-of-range*, *overlap*}. Different versions of code can lead to cut and paste errors if programmers are not careful when fixing bugs.
- Register pressure is increased due to range check variables required by this DASk.
- Extra branches are required to handle loops and range checks. In particular, the while loop used to loop over the full rows of data is called once per data block. Fortunately, my DASk implementation is written to avoid divergent branching.
- Performance is sensitive to the initial grid size. If the grid size chosen by the programmer is not an even multiple of the actual *workLoad* (#SMs * #Blocks), then performance suffers as a few SM cores process the extra rows while the rest of the SM cores sit idle.
- Coherence is reduced since the cooperative stride across all data blocks results in each new data block being a full data row apart from the previous data block. This hinders caching by the memory controllers when moving row to row.

Row DASk:

My *Row* DASk (shown in Figure 10.4) follows a 2D layout where *m* thread blocks are load balanced across *r* fixed rows into *c* columns.

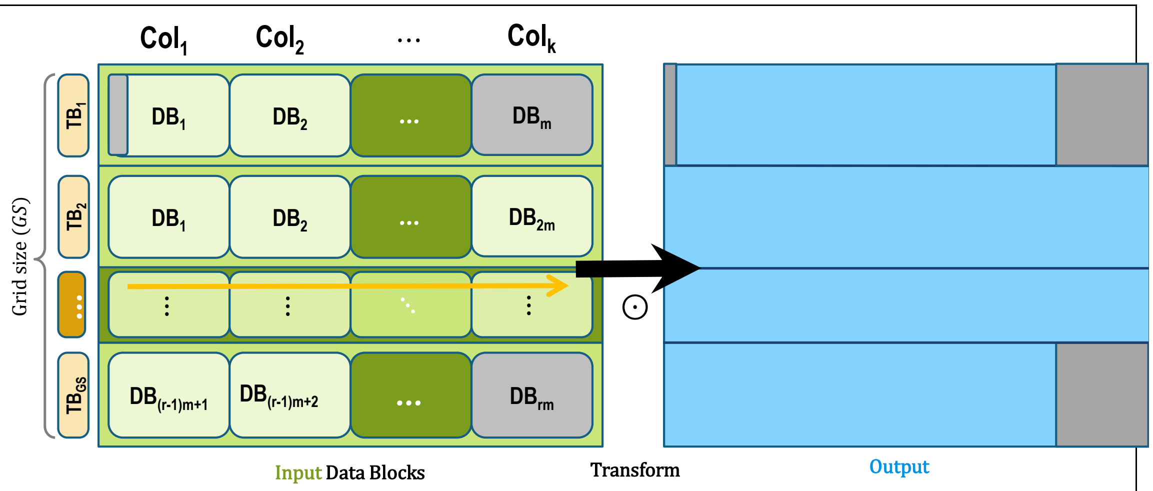


Figure 10.4: The *Row* DASk layout.

As discussed in Section 5.4, the Row DASk is quite similar to the Column DASk. It also uses a two dimensional grid layout. However, the rows and columns are reversed. For the Row DASk, the original input data (*n*) is grouped into *m* fixed-size data blocks (*DBS*) with the data blocks being laid out on a 2D grid, with *m* = $\lceil n/DBS \rceil$. The number of rows (*R*= *grid.height*) is chosen ahead of time by GPU programmers as a static fixed-size constant. The number of columns (*C* = *data blocks per row*) varies with *m* (and thus *n*) in order to fully cover all input data, with *C* = $\lceil m/R \rceil$. However, for better load balancing of work across thread blocks, my actual computation of the number of columns per row is more complex (see Section 5.6.3). The Row DASk maps one thread block onto each data row, with each thread block being responsible for processing all data blocks along its assigned row.

The orientation change induces several other differences between the Row and Column DASk in alignment, range checks, and load balancing. To better support coalescence, as discussed in Section 5.6.1, my Row DASk aligns the input data to a data warp boundary before marching down the first row (the Block and Column DASks do not warp align data; so I/O performance may suffer if the user passes in unaligned data arrays). Supporting warp-aligned data implies that the first data warp in the first data block in the first row may be partially empty, which requires careful range checks. Similarly, the last data block in the last data row may also be partially empty, which also requires careful range checks. To help mitigate underutilization, I use a complex load-balancing scheme, as discussed in Section 5.6.3, to distribute the data blocks approximately evenly across the thread blocks representing each data row. With a difference of at most one data block in length between any two given rows. I bias the load balancing against the thread blocks (rows) containing the first data block and last data block as they may require more expensive range checks when processing those data blocks.

As discussed in Section 5.6.2, I also introduced the ‹FIRST?› ‹MIDDLE*› ‹LAST?› range check pattern for use with the Row DASk. I use regular expression notation to indicate how many data blocks are involved in each group (? = zero or one data blocks, * means zero or more data blocks). This pattern removes range checks out of the large excluded middle and pushes required range checks into only the first and last data blocks. There is also a special case ‹BOTH› for when the first and last data blocks are one and the same. This approach amortizes the required per-element range checks in the first and last data blocks across the entire data set of *m* data blocks.

- ‹FIRST?›: As a result of warp-aligning the data array to a warp boundary, the first data block may require partial [*start*, …) range checks on the first data warp.
- ‹LAST?›: As a result of using fixed size data blocks, the very last data block may be only partially full and thus require partial (…, *stop*] range checks on all data warps.
- ‹MIDDLE*›: The data blocks in the vast excluded middle do not require range checks.
- ‹BOTH?›: For very small data sets, it is possible that the ‹FIRST? › and ‹LAST?› data blocks are one and the same. This case requires full [*start*, *stop*] range checks.

My Row DASk has five main benefits and five limitations.

Benefits:

- To better support coalescence, this DASk warp-aligns data access (input/output) to a warp boundary [0, 32, 64, …].

- For better SM core processor utilization, this DASk load balances work as evenly as possible across thread blocks (rows) while also biasing against the first and last range checked blocks.
- For better performance, this DASk amortizes range checks using the ‹FIRST?› ‹MIDDLE*› ‹LAST?› pattern, which pushes range checks into the first and last data blocks and allows any required range checks to be amortized across the entire data row.
- This DASk can support algorithms that require ordered data for correct results by supporting a sequential access pattern as each thread block marches along its data row, block by block. However, GPU programmers still need to ensure sequential access within each data block for correct overall behavior.
- For better throughput, this DASk supports an efficient access pattern into memory, using a 2D grid layout.

Disadvantages:

- High setup costs (warp alignment, load balancing, range checks, …) need to be amortized across *R* data blocks along each row.
- GPU programmers need to write four slightly different versions of the same code to handle the four different range check cases {*Both*, *First*, *Middle*, *Last*}. These different versions of code can lead to cut and paste errors if programmers are not careful when fixing bugs.
- This DASk can increase register pressure due to setup costs, load balancing and range check variables used by this DASk.
- Extra branches are needed to handle the four range check cases {*BOTH*, *FIRST*, *MIDDLE*, *LAST*}. In particular, the while loop used for the ‹*MIDDLE**› case gets called once per data block. Fortunately, my DASk implementation was written to avoid divergent branching.
- Performance is sensitive to the initial grid size. If the grid size chosen by the programmer is not an even multiple of the actual *workLoad* (= *nSMs* × *nConBlocks*), then performance suffers as some SMs process a few extra data blocks along the row while the rest of the SMs idle.

DASk Summary:

All three DASks support high performance parallel programming. The Copy case study in Chapter 4 and 5 showcases the Block DASk. The Histogram case study in Chapter 8 showcases the Column DASk. The Reduce/Scan case studies in Chapter 6 and the Radix Sort case study in Chapter 9 showcases the Row DASk.

Taking advantage of both instruction-level parallelism and thread-level parallelism on GPUs is essential to unlocking high performance. All three DASks are written to take two template parameters ‹*nWarps*, *nWork*› which allow experiments on ILP and TLP. This approach allows programmers to find the optimal balance of ILP and TLP for best performance for a given problem.

10.2 Summary and Lessons Learned:

GPUs unlock massive amounts of parallelism for programmers to take advantage of. Writing parallel code that is correct and high performing is quite difficult. My three data access skeletons help solve many of the issues that GPU programmers must grapple with and eases the burden of implementing their own GPU programs. These issues include mapping threads onto data using the 2-level CTA hierarchy, coalescence, bank conflicts, and multi-issue. To achieve high performance, GPU programmers also need to take advantage of ILP and TLP techniques. My various case studies show useful GPU primitives implemented on top of my three DASks. These primitives achieve high performance results using varying amounts of ILP and TLP.

Lessons Learned:

Summarized here are some of the key lessons on high performance GPU programming that my dissertation has presented.

- Use my three data access skeletons (Block, Column, or Row) as starting points to speed-up the writing of your own GPU kernels.
    - The 1D Block DASk is a good choice for simple kernels such as Map operations (such as Fill, Copy, Scatter, Gather, …) due to its simplicity and coherent access pattern.
    - The 2D Column DASk is a good choice for complex operations on unordered data (such as Histogram, Reduce, …).
    - The 2D Row DASk is a good choice for complex operations on ordered data (such as Scan, Radix Sort, …).
- Partition data across the 2-level CTA hierarchy
    - Partitioning can be implemented in a deterministic lock-free manner provided that the data is known a priori and remains static during the algorithm.
    - Programmers should partition data into *m* fixed-size data blocks
        - *TBS* = *nWarps***WarpSize*
        - *DBS* = *nWork***TBS*
        - $m = \lceil n/DBS \rceil$
    - Data blocks represent the bottom level of the CTA hierarchy (threads within a thread block).
        - BASks (Block-by-Block and Warp-by-Warp) are a good starting point for loading inputs and storing outputs for processing each data block
        - Programmers still need to design data layout and write kernel code that solves their specific problem.
    - DASks handle the top level of the CTA hierarchy (thread blocks within a grid).
        - Block DASk – one thread block per data block (*GridSize* = *m*)
            - 1D layout
            - Last block may be partially full requiring range checks
        - Column DASk – distribute *m* data blocks evenly across *C* columns
            - 2D layout
            - *C* = *GridSize.y* (picked by programmer), $C \leq 1000$.
            - $R = \lceil m/C \rceil$ = number of blocks per column
            - Each of *C* thread blocks processes *R* data blocks along its assigned data column
            - Last row may be partially full requiring range checks.
        - Row DAsk – distribute *m* data blocks evenly across *R* rows
            - 2D layout
            - *R* = *GridSize.x* (picked by programmer), $R \leq 1000$.
            - $C = \lceil m/R \rceil$ = number of blocks per row

        - Each of *R* thread blocks processes *C* data blocks along its assigned data row.
        - First and last data blocks may be partially full requiring range checks.
- Use ILP and TLP techniques to increase parallel performance
    - Unlock pipelined performance via ILP by working on ‹*nWork*=[1-8]› work items per thread. Regroup and reorder instructions using loop unrolling and software pipelining to mitigate instruction dependencies and reduce pipeline stalls.
    - Unlock multi-threaded parallel performance by launching ‹*nWarps*=[1-8]› thread warps per thread block.
    - Perform experiments on ‹*nWarps*, *nWork*› pairs to find the optimal performance.
    - The optimal balance between ILP and TLP varies across problems, implementations, and platforms due to varying constraints on occupancy.
- Mitigate constraints on occupancy to increase parallel performance.
    - Redesign memory layouts to reduce shared memory usage to increase occupancy
    - Rewrite code to reduce register usage to increase occupancy.
    - Choose a grid size that is a multiple of the concurrent work load so that thread blocks divide evenly into the number of SM cores on a specific GPU card.

APPENDIX A: GLOSSARY

Absolute speedup: *Speedup* in which the time to run the best *parallel* solution is compared to the time to run the best *serial* solution to the same problem, even if they are using different algorithms.

Abstract Data type: An abstract data type (ADT) is a model for data structures or data types that can be described by their behavior or semantics. An abstract data type is defined only by the operations that may be performed on it; pre-conditions on the inputs to those operations; post-conditions on the outputs from those operations; and possible descriptions of the required time or space complexity of those operations using asymptotic notation. For example: A *stack* can be defined by 3 operations – `push`, `pop`, and `top`, each taking constant $O(1)$ time. See also *object-oriented programming*.

Abstraction: In computer science, abstraction manages complexity. A certain level of complexity is established on how a person interacts with the computing system, while more complex details below the current level are suppressed. Thus a programmer works with an idealized well-defined interface and can add more levels of functionality later that would otherwise be too complex to handle. There are two main types of abstraction – control and data. *Control abstraction* involves using sub-programs or sub-routines to manage the control flow through a program. *Data abstraction* allows the grouping of data bits into meaningful chunks, i.e. data types, records, structures, etc. *Object-oriented* classes combine abstractions of both functionality and data.

Algebraic Simplification: A *data-flow optimization* technique that uses algebraic laws to simplify or re-order instructions for better performance. See also *constant folding*, *code fusion*, and *strength reduction*.

Algorithm: An algorithm is a step-by-step recipe to perform a task that solves a problem – an unambiguous set of instructions that performs the task by operating on input and producing output. An algorithm is typically expressed using language-agnostic *pseudo-code*, which can be implemented in any programming language.

Algorithmic Skeleton: Synonym for *pattern*, specifically the subclass of patterns having to do with algorithms.

Algorithm strategy pattern: A class of patterns that emphasize the parallelization of the internal workings of algorithms.

Aliasing: Two similar definitions: 1) When two pointers (or expressions that resolve to memory addresses) refer to overlapping memory locations. For example, if pointers p, q both point to the exact same location, then $p[0]$ and $q[0]$ alias each other. 2) When two different variables are mapped onto the same physical register by scheduling (static compiler or dynamic hardware). This can result in *write after read* (*WAR*) or *write after write* (*WAW*) dependencies through the aliased register.

Alignment: With alignment, data starts on some fixed boundary, where the boundary is a multiple of some integer number, usually measured in bytes. For instance 64-bit integers are aligned to start on 8 byte boundaries within memory (0, 8, 16, 24, …). See also *data alignment*, *cache-line alignment*, *warp-line alignment* and *coalescence*.

All-*k*NN: See *All k-Nearest Neighbor*.

All *k*-Nearest Neighbor: A type of *Nearest Neighbor Search* (All-*k*NN) where each query point in a query set *Q* is matched to the *k* closest points in a search set *S* under some distance metric. In this search, the query set *Q* and search set *S* are one and the same (*Q* == *S*). See also *k*d-tree.

All-NN: See *All Nearest Neighbor*.

All Nearest Neighbor: A type of *Nearest Neighbor Search* (All-NN) where each query point in a query set *Q* is matched to the closest point (*k* = 1) in a search set *S* under some distance metric. In this search, the query set *Q* and search set *S* are one and the same (*Q* == *S*). See also *kd-tree*.

All-to-All: A special case of a mapping relationship where every source object is related to every destination object in a set, table, or array of parallel objects. See also *cardinality*.

ALU: An ALU (arithmetic logic unit) is a small processing unit which can perform simple computations such as addition or multiplication on integer data.

Amdahl's Law: *Speedup* is limited by the serial portion of the total work. Given α and p, where $\alpha \in [0,1]$ is the percentage of the program which is *serial* and p is the number of *parallel* processing cores used to process the parallel portion of the program. Then the maximum speedup according to Amdahl's Law is $S(\alpha, P) = \frac{1}{\alpha + \frac{1-\alpha}{p}}$. Compare with *Work+Depth Analysis* or *Gustafon's Law*.

AMP: Microsoft's C++ Accelerated Massive Parallelism (AMP) is a native programming model that extends the C++ programming language and its associated runtime library to enable data-parallel programs on data-parallel hardware, such as GPUs.

ANN: See *Approximate Nearest Neighbor*.

Anti-Dependency: See *write after read* (*WAR*).

Analysis of Algorithms: See *Asymptotic Analysis*.

Application Programming Interface (API): An interface is the set of function calls, operators, and/or classes through which an application developer accesses a predefined module or library to accomplish some useful task(s). Ideally *information hiding* is used to hide the implementation details of the module while allowing functionality to be accessed through the API's interface.

Approximate Nearest Neighbor: A type of *Nearest Neighbor Search* (ANN) where each query point in a query set *Q* is approximately matched to the closest point in a search set *S* under some distance metric. In ANN searches, each answer is approximately correct with high probability. However, there is always a small chance that another point (the true solution) is even closer. ANN searches usually run faster than other types of NN searches, but at the cost of some incorrect answers in the search results.

Arithmetic Intensity: The ratio of computational to communication operations in a program. Comparing the programs arithmetic intensity with the hardware's theoretical arithmetic intensity helps decide whether computation or communication limits performance.

Arithmetic Logic Unit: See ALU.

Array of Structures (AoS): A data layout for collections of data elements where all of the heterogeneous data fields for a single element are stored in adjacent physical memory locations. Compare with *Structure of Arrays (SoA)*.

Associative Cache: A k-way cache organization in which different data from different locations in main memory are mapped onto the same small subset of physical locations within a cache (4 cache-lines are shared in a 4-way associative cache). A fully associative cache allows data from anywhere in main memory to be stored anywhere in cache memory. Older data only needs to be evicted from cache when the entire associate k-way set (or entire cache for fully associative) becomes full. Contrast with *direct mapped cache*.

Associative Operation: An operation $\otimes$ is associative, if $(a \otimes b) \otimes c = a \otimes (b \otimes c)$ for all a, b, and c in its domain. Associative operations allow data to be re-grouped but no re-ordered. Integer addition is associative for both mathematics and computation. However, even though mathematical addition for reals is associative, floating-point addition on a computer is not due to truncation and round-off errors (Press et al, 2007). If these small errors can be lived with then floating-point math can be considered approximately associative (pseudo-associative). Some techniques for parallelizing scans and other ordered operations require associativity for correct results.

Asynchronous Coordination: Involves multiple threads (or tasks) trying to communicate data or coordinate work between themselves. Asynchronous coordination allows threads to transfer data independently from each other. One thread can start a data transfer and then start doing other useful work while waiting for the data transfer to complete. Interleaving computation with coordination between threads allows *latency hiding* to improve overall performance. Contrast with *synchronous coordination*.

Asymptotic Analysis: AKA Analysis of algorithms - Measures the overall intrinsic efficiency of algorithms without getting bogged down in the actual details of hardware, software, or implementation. Two main principles are used 1) Ignore machine-dependent or implementation-dependent constants; 2) Look at the overall growth rate of resources (time, space, etc.) as the input size (n) goes to infinity ($n \to \infty$). See also *Asympotic Complexity*.

Asymptotic Complexity: A limit on resource usage, including work, time and space but also for ratios such as *speedup* and *efficiency*. See also *Big 'O' notation*, *Big 'omega' notation*, *Big 'theta' notation*, *Little 'O' notation*, and *little 'omega' notation*.

Asymptotic Running Time	**Description**
$\theta(1)$	Constant Time
$\theta(\log n)$	Logarithmic Time
$\theta(\log^k n)$	Polylogarithmic Time
$\theta(n)$	Linear Time
$\theta(n \log n)$	Log-Linear or Linearithmic Time
$\theta(n^2)$	Quadratic Time
$\theta(n^k)$, $k > 1$	Polynomial Time
$\theta(k^n), k > 1$	Exponential Time

Asymptotic Efficiency: An *asymptotic complexity* measure for *efficiency*.

Asymptotic Speedup: An *asymptotic complexity* measure for *speedup*.

Atomic Instruction: See *atomic operation*.

Atomic Operation: A small set of instructions guaranteed to appear as if it occurred as a single indivisible instruction by the rest of the system. Once the atomic operation has started other threads cannot interrupt the current thread until the atomic operation has completed. See also *lock, thread-safe, mutual exclusion, non-blocking algorithm, lock-free algorithm*, and *wait-free algorithm*.

Atomics: The set of *atomic operations* supported on a specific machine architecture. Modern atomic operations are based on *compare-and-swap* semantics. Older architectures used *test-and-set*, or *fetch-and-add* semantics.

Atomic Scatter Pattern: A non-deterministic data pattern in which multiple competing writers to a single memory location result in exactly one value being written and all other values being discarded. The value written is chosen non-deterministically from the multiple competing sources. The only guarantee is that the final written value will belong to at least one of the competing writers. See also the *PRAM machine model*.

Auto-tuning: Auto-tuning is an automated process which adjusts and selects parameters in parameterized code in order to find the set of parameters which result in the best performance.

Back-Tracking: Backtracking is a general algorithmic approach used to find all (or some) solutions to computational problems. Typically a tree or graph is explored or built as part of finding these solutions. The back-tracking algorithm incrementally builds partial solutions and stores other possible candidate solutions (unexplored paths) encountered along the way in some simple data structure (queue or stack). It abandons each partial solutions as soon as it determines that they cannot possibly be part of the valid solution set. It then backtracks to any other candidate solutions kept in storage until all possible candidates (solution paths) have been exhausted (the storage data structure is empty).

Balanced: A balanced tree with *n* nodes has a maximum tree height of $\lceil \log_2 n \rceil$, in other words, there is a difference in path length from the root to any leaf node of at most one level across all leaf nodes.

Bandwidth: The rate at which information is transferred. This can be between the CPU and memory or over some other communications channel. The amount of data that is communicated per unit of time, usually measured in megabytes per second (MB/s) or gigabytes per second (GB/s). See also *throughput*.

Bank Conflicts: A bank conflict occurs when multiple threads attempt to access different data elements stored in the same memory bank at the same time. The GPU hardware scheduler solves this resource contention by serializing access to the memory bank which negatively impacts I/O performance.

Banked Memory: With banked memory, also known as interleaved memory, we have multiple parallel sections of memory with each bank represented by its own memory chip (or separate section on a chip). Memory accesses return *k* values from *k* banks in parallel increasing memory throughput by a factor of *k* for coherent memory accesses. Banked memory is one way to compensate for the relatively slow speed of RAM memory compared to CPU registers. GPU hardware uses banked memory for both shared and global memory to greatly increase aggregate memory throughput.

Banking: Increases memory transfer throughput by having multiple banks of memory that are accessed in parallel. Banks usually work modulo the bank size. A memory system with *k* banks can be up to *k*× faster than the equivalent non-banked memory solution.

Barrier: A check-point that each thread must wait at until all threads reach this check-point before any thread proceeds past the check-point. This allows the coordination of execution across threads within a thread block on GPUs.

Barrier Synchronization: A multi-threaded computation is often divided into multiple phases. Often all threads must complete one phase fully before any thread is allowed to move onto the next phase for correct program behavior. A barrier is a form of synchronization that ensures this behavior. Threads arriving at the barrier wait there until all threads have arrived, at which point all threads are allowed to continue. Barriers can be implemented using *atomic operations*. Barriers do *serialize* thread behavior which slows parallel performance.

BASk: See *block access skeleton*.

Batching: Batching helps reduce *register pressure* in GPU kernels by grouping partially unrolled work items into fixed sized batches. Typically the amount of batched work is in the range of [2..4] work items.

BFS: See *Breadth-First-Search*.

Big 'O' Notation: Complexity notation that denotes an asymptotic upper bound on resource usage; written as $O(g(n))$. Big 'O' notation is useful for comparing the algorithmic efficiency of different algorithms. $O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$ $\}$. See also *asymptotic complexity*, *little 'o' notation*, *big 'omega' notation*, *little 'omega' notation*, and *big 'theta' notation*.

Big 'Omega' Notation: Complexity notation that denotes an asymptotic lower bound on resource usage; written as $\Omega(g(n))$. Big 'Omega' notation is useful for comparing the algorithmic efficiency of different algorithms. $\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq c \cdot g(n) \leq f(n)$ for all $n \geq n_0$ $\}$. See also *asymptotic complexity*, *big 'O' notation*, *little 'o' notation*, *little 'omega' notation*, and *big 'theta' notation*.

Big 'Theta' Notation: Complexity notation that denotes an asymptotic upper and lower bound on resource usage; written as $\Theta(g(n))$. Big 'Theta' notation is useful for comparing the algorithmic efficiency of different algorithms. $\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2 and n_0 such that $0 \leq c_1 \cdot g(n) \leq$

$f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0$ }. See also *asymptotic complexity*, *big 'O' notation*, *little 'o' notation*, *big 'Omega' notation*, and *little 'omega' notation*.

Binary Tree: A tree data structure composed of binary nodes where each node contains a data element with at most two child links {*left*, *right*} which point to the left and right child nodes in the tree.

Binary Heap: A binary heap is a *complete binary tree* which satisfies the heap ordering property. Either min-heap (The key value of each node is greater than or equal to the value of its parent, with the minimum-key element kept at the root) or max-heap (the key value of each node is less than or equal to the value of its parent, with the maximum key element kept at the root).

Binning: The process of partitioning labeled data into a collection of labelled sub-collections. Each labelled sub-collection only contains data belonging to that specific label.

Bin Pattern: A generalized version of the *split pattern*, which in turn is a generalization of the *pack pattern*. The bin pattern takes as input a collection of data and a collection of labels to go with every element of that collection. This pattern reorganizes data into labelled sub-categories (bins) for every unique label in the input. The *deterministic* version of this pattern is *stable*, in that it preserves the original order of the input for data with the same label. Typically used in *radix sort* and other binning algorithms.

Bit-Level Parallelism: Bit-level parallelism is a well-known programming technique for increasing performance which can be thought of as a special form of *vector-level parallelism*. Bit-level parallelism, also known as word parallelism and SWAR (SIMD within a Register), works on multiple small data types in parallel using a single large word instruction (typically in the machines native hardware word size). For instance, a 32-bit hardware register can store four 8-bit *bytes*, two 16-bit *words*, or one 32-bit *dword*. Programmers that work conceptually with bytes or words but process data using 32-bit instructions can potentially work on four or two data items at once using standard machine instructions. This allows a potential speed-up of 4× or 2× respectively.

Block: Three definitions: 1) A state in which a thread cannot proceed while it waits on some other system resource or synchronization event. 2) A 3D sub-region of memory from a larger data collection, see also *run* and *tile*. 3) A CUDA thread block of a fixed shape (1D, 2D, or 3D) and size. See also *CTA*.

Block Access Skeleton: A *block access skeleton* (BASk) is a machine dependent memory access pattern (or algorithm) for GPUs. For instance, the Block by Block BASk accesses and covers an entire data block by supporting the bottom level of the 2-level CTA mapping. See also *data access skeleton*, *Block-by-Block* and *Warp-by-Warp BASks*.

Block-by-Block BASk: An efficient access pattern at the bottom level of the 2-level CTA mapping that supports coalescence. Each thread within a thread block of size TBS accesses its own unique data element within the current group of TBS elements and then strides (*stride* = *TBS*) to the next group of TBS data elements within the data block to access. See also *block access skeleton*. Compare with the *Warp-by-Warp BASk*.

Branch and Bound Pattern: A *non-deterministic* pattern meant to find one satisfactory answer when many possible answers exist. The term *branch* refers to using concurrency. The term *bound* refers to limiting the computation in some manner. This pattern is often used to implement an efficient *search* pattern.

Branch Condition: An expression that resolves to a {*true*, *false*} predicate which is then used to choose between two paths of execution within a section of code, function, or program.

Branch Divergence: Because all threads within a thread warp step through an instruction stream in lockstep, branch divergence occurs when some threads within a warp take the *true* branch path and the rest of the threads take the *false* branch path. GPU hardware currently handles branch divergence via "serialization by predication". *Serialization* mean that the code for both branch paths are executed by all threads. *Predication* means that individual threads within each warp are enabled/disabled via predicate bit masks as appropriate for the code represented by each branch. This solution guarantees correct per-thread branch outcomes at the cost of reduced performance. In the worst case, where branches are nested five levels deep, with each thread taking its own unique path, branch divergence can cause parallel threads to run up to 32× slower than equivalent single-threaded code which follows only one taken path through the branch tree.

Branch Prediction: Branch prediction is an entire family of *ILP* optimization techniques meant to reduce the *branch delay* caused by *control hazards*. Branch prediction uses special logical circuitry to try to predict which branch {*true*, *false*} a given branch will take and will either start pre-fetching instructions from the predicted path or start executing speculative instructions from the predicted path. If the guess is correct than the core starts executing the prefetched instructions which reduces the cost of branch delays or commits the speculative instructions which avoids the cost of branch delays. If the guess is incorrect than either the pre-fetched instructions are ignored or the speculative instructions and any side effects from the incorrect path need to be *canceled* and the instruction counter reset to the correct path. See also *Branch Speculation*.

Branch Selection: see *selection pattern*.

Branch Speculation: Branch speculation is an entire family of ILP techniques meant to reduce the *branch delays* caused by *control hazards*. With branch speculation, the architecture has enough functional resources to execute speculative instructions from both paths {*true*, *false*}. Once the branch outcome is known the incorrect path's speculative instructions & any side effects must be *canceled* and the correct path's instructions are committed. This approach avoids branch delays at the cost of extra functional units and extra control logic to commit or cancel speculative instructions. Nested branch instructions can still cause branch delays as it is infeasible in practice to speculate along all possible branching sub-paths. See also *Branch prediction*.

Breadth-First-Search: An algorithm for searching a tree or graph data structure. The algorithm starts at the root (or an arbitrary starting node) and explores all neighbor nodes first before moving to the next level of neighbors (neighbors of neighbors). Nodes to be visited are typically stored using a *Deque* in *FIFO* order. See also *Depth-First-Search*.

Broadcast Model: A *coordination* model, where one thread (or task) sends the same message to all other threads (or tasks) in the same group. See also *remote message passing*.

Brute-Force Search: A geometric search where each point in a query set Q (with m points) is compared to all points in a search set S (with n points) to find the k closest neighbors. This typically takes $O(mn)$ time, or quadratic time $O(n^2)$ time if $S == Q$. See also *nearest neighbor search*.

BSP Tree: A binary space partitioning (BSP) tree is a hierarchal *spatial partitioning data structure* used to speed up geometric searches. Each tree node typically has two children. BSP trees typically are used to partition *d*-dimensional space by recursively dividing the starting bounding object into 2 smaller geometric regions separated by a *d*-dimensional hyperplane. See also *BSP tree*, *octree*, and *kd-tree*.

Buffering: A buffer is a fixed-size region of memory storage used to temporarily store data while it is being moved from a producer to a consumer. Buffering is often used to move data between RAM memory and files on disk drives. Buffering is often used when moving data between *processes* within a computer, for instance when implementing the *pipeline pattern* or a *producer-consumer* relationship. Buffering introduces issues with stalling the producer when the buffer is full and stalling the consumer when the buffer is empty.

Busy Waiting: A form of mutual exclusion, where the current thread tries repeatedly to acquire a lock, this busy waiting is typically done inside a small tight loop (such as a spin-lock). The assumption with this style of programming is that there is not much resource contention for the lock and that the code/data protected by the lock will be processed quickly by the primary thread that successfully acquires the lock.

By Pointer: A parameter to a function that is passed in indirectly as a memory address which needs to be dereferenced in order to access the true value. Allows the original value to be changed as part of the functions execution. For good or bad, the pointer address can be manipulated to access data elements other than the one originally pointed to. See also *by reference*, *by value*, and *parameter*.

By Reference: A parameter to a function that acts exactly as if it were the original location passed into the function. Allows read/write access to the original element only. The original element must exist before passing the reference into a function. See also *by pointer*, *by value*, and *parameter*.

By Value: A parameter to a function that is a copy of the original value passed into the function. The original value will remain unchanged regardless of what changes are made to the copy's value during the functions execution. See also *by pointer*, *by reference*, and *parameter*.

Bypassing: An ILP optimization technique where extra shortcut circuitry is added into the instruction pipeline to get intermediate output results from one stage into another stage where the results are needed as inputs. Bypassing reduces the instruction delay (number of stalls) between two dependent instructions in the pipeline at the cost of a more complex processing core.

Byte Stream: A byte stream is a stream of data in which the individual machine bits are grouped into units of 8-bits called bytes. Byte streams are also used as a basic abstraction for how to communicate data between two processing units. One unit writes bytes into the communications channel and the other unit reads them. Byte streams are often used to connect producers with consumers (see *producer-consumer relationship)* as used in the *pipeline pattern*. *Buffering* is often used to smooth out performance differences between the producer and consumer. On most operating systems, access to any file is done as a byte stream.

Cache: A part of the memory system that stores copies of data temporarily in faster memory so that future requests for that same data can be handled more quickly. Caches are typically implemented in hardware with support from the operating system and thus hidden from the programmer. Caches are designed to enhance average memory access speeds by exploiting temporal and spatial locality. Caches in modern computers are multi-level. *Cache coherence* is required to avoid data errors when caching. See also *cache-aware* and *cache-oblivious programming*.

Cache-Aware Programming: Programs that are written to take advantage of the multi-level memory hierarchy on a specific machine architecture for better I/O performance. Programmers learn the actual details of cache-line sizes and page sizes for a specific memory architecture. Programmers then use that knowledge to write code that transfers data in an *aligned*, *coherent*, and *fully-loaded* manner that uses memory more efficiently. Since cache-aware programs are tied to a specific memory architecture, they are not portable and need to be carefully tweaked on different machines in order to run well. Contrast with *cache-oblivious programming* and *streaming algorithms*.

Cache Alignment: When memory accesses are aligned to start on cache-line boundaries (0, 64, 128, 192, 256), memory accesses tend to run faster.

Cache Coherence: A mechanism for keeping multiple copies of the same data stored in different caches consistent. In other words, all threads accessing the same original memory location will see the same value despite there being multiple copies scattered throughout the multi-level cache system.

Cache Coherent Architecture: Refers to the consistency of data stored in multiple separate caches for parallel computing units. A multi-core system may have a L1 cache that is private to each core but an L2 cache or RAM main memory that is shared across all the cores in the system. As data is accessed, each core may cache its own copy of an original data element in its L1 cache. When one core updates its copy of data then all other cores will have stale copies (incorrect) in their caches. A cache coherent memory architecture will ensure that all processors receive the most up-to-data version of any data regardless of where it is stored (L1 cache, L2 cache, RAM memory).

Cache-conflict: When multiple locations in memory are mapped onto the same location in cache memory only a subset of them can be kept in cache.

Cache Fusion: A memory access optimization where data is partitioned into small tiles and vector operations are executed on each tile so that intermediate values can be kept in cache for better performance.

Cache Line: The fixed unit sizes in which data is retrieved and held by a cache; the cache line size is typically 8-64 bytes on CPU architectures and typically 128 bytes on modern GPU architectures. Larger cache lines allow for more efficient bulk transfers but also increase the inefficiency of incoherent memory accesses.

Cache Miss: A cache miss is caused when a requested instruction or data operand(s) is not immediately available from the fastest level of a memory hierarchy and the entire instruction processing pipeline must suspend operation until the cache memory is filled with the requested instructions or data via memory I/O and then resumes operation. There are four main types of cache misses – *Compulsory*, *Capacity*, *Conflict*, and *Coherency*. Capacity, Conflict, and Coherency misses can lead to *cache thrashing*.

Cache Miss Rate: The fraction of memory cache accesses that result in a cache miss.

Cache-Oblivious programming: Refers to writing programs that have good cache behavior without knowing the actual size or design of a specific memory architecture (caching system) in advance. This is typically accomplished by recursively partitioning data into smaller and smaller processing chunks until at some point a chunk becomes small enough to fit into a single *cache-line* or page at some level of the *memory hierarchy*. Cache-oblivious algorithms are portable across different memory architectures as they do not depend on any particular memory details. Contrast with *cache-aware* and *Byte stream* programming.

Cache Thrashing: Cache thrashing is when a sequential (or parallel program) spends most of its time reloading cache blocks into the cache and waiting on the cache loads to occur instead of getting useful work done. This is caused because a small working set of cache blocks all get mapped onto the same cache line in cache memory and the system keeps evicting and then reloading these competing cache blocks as the program progresses. *Capacity*, *conflict*, and *coherency* cache misses are the typical cause of cache thrashing.

Caching: A memory organization that exploits temporal and spatial locality in instructions and data for better amortized memory access speeds for frequently reused instructions or data.

Cancellation: Two definitions: 1) Refers to hardware invalidating all instructions currently in the instruction pipeline after the cancelled instruction in response to a hardware exception, speculative execution, or other event. Cancelled instructions are similar in behavior to *NOP* instructions. 2) The ability to stop a running task (thread) from another task (thread).

Capacity Cache Miss: A capacity miss occurs when a cache simply cannot contain all the cache blocks needed during the full execution of a program. Cache blocks need to be discarded from the cache to make room for other cache blocks needed by the program. The discarded cache blocks may themselves need to be reloaded later as the program continues to execute.

Cardinality: A mapping relationship between two different sets, tables, or arrays of parallel objects. For instance data and processing cores. The main relationship types are *one-to-one*, *one-to-many*, *many-to-one*, and *many-to-many*. With some special case relationships such as *all-to-all*, *one-to-several* and *several-to-one*.

Category Reduction Pattern: A combination of search and segmented reduction, each data input has a label and reduction occurs only between elements with the same label. The output is a set of reduced results for each label. The map-reduce model is the classic example of this pattern.

Ceiling Function: Given any real number x, there exists a unique integer n such that $n < x \leq n+1$. We say that $n+1$ is the ceiling of x, often denoted as $\lceil x \rceil = n+1$. See also *floor function*.

Chosen Digit: For *Radix Sort*, Given a key taken from a large numeric range [0, *m*) represented by a string of up to *k* digits, with each digit from a small range [0, *d*) then the radix sort will take *k* sorting passes. Each pass uses a *stable* sort such as *Counting Sort* to distribute *n* keys into *d* sorted *runs*. The number of digits (or sorting *passes*) can be computed as $k = \lceil \log_d m \rceil$. The *chosen digit* represents the current digit, $j \in [0, k)$, within the key used to perform the current pass's stable sort. The *k* passes proceed from the least to most significant digit in the key for a LSD radix sort.

Cilk: Cilk (and Cilk++ and Cilk Plus) are general purpose programming languages designed for multi-threaded parallel computing. They extend the C and C++ programming languages with constructs to express parallel loops and the fork-join parallel pattern. Originally developed by MIT and currently owned by Intel.

Class: A class is an expanded concept of a data structure; instead of holding only data, it can hold both data and the functions intended to manipulate that data. An *object* is a specific concrete instantiation of a class. See also *object-oriented programming*.

Cloud: A set of computers, typically kept in a data center that can be allocated dynamically and accessed remotely. Unlike a *cluster*, cloud computers are typically managed by a third party and may host multiple applications from different, unrelated users.

Cluster: A set of computers with distributed memory communicating over a high-speed interconnect. The individual computers within a cluster are often called *nodes*.

Coarse-Grained Multi-threading: With coarse-grained multi-threading, *context switches* between active threads only happen when the current thread is likely to *stall* for a long time or a large time-slice (measured in milli-seconds) has gone by. Thread contexts tend to be *heavy-weight*. As a result, this approach cannot hide *latency* for short term pipeline stalls caused by various hazards.

Coarse-Grained Parallelism: With coarse-grained parallelism, the ratio of coordination to computation events in a parallel algorithm is quite large. In other words, coordination & communication events occur infrequently.

Code Fusion: 2 similar definitions: 1) An optimization for a sequence of operations on parallel data each with their own separate input/output phases into a single fused operation with a single input/output phase. This optimization reduces the total number of I/Os required to process data. 2) This *data-flow optimization* technique combines several operations with one simpler equivalent operation.

Copy: A type of *map* primitive, which copies a source array S of n elements into a destination array D, with no overlap allowed. Also see *Fill*, *Scatter*, and *Gather*.

Closest Heap Data Structure: A special compound data structure used with *kNN* and *All-kNN nearest neighbor searches* to return the k nearest neighbors. This compound data structure first acts like a simple array of k elements and then later like a fixed-size *binary heap* of k elements. The first k-1 search results are appended into the array. After adding the k^{th} search node into the array, the array is converted into a max-distance heap, taking $O(k)$ time to run. Subsequent search point results encountered during the NN search are then compared against the top point in this closest heap data structure. If the new distance is closer, the top is replaced by the new point. The correct heap ordering is then restored by demoting the new top which takes logarithmic time $O(\log k)$ in the worst case to run.

Coalescence: With *coalescence*, The GPU memory controllers support accessing an entire warp (32 threads) of 32 (32-bit) data elements (128 bytes) in parallel. In fact, 128 bytes is the standard I/O access size. Thus each memory controller can read or write up to 32 elements in parallel for the cost of a single I/O. Similar to CPU cache-lines, the aggregate data access must be aligned to a warp-line boundary (some multiple of 128 bytes). The memory controller has native "gather" support so each thread can access any relative index [0..31] within a warp-line at no extra cost. However, each thread's access index must be unique otherwise the hardware will serialize access across threads competing for the same address. This means the memory access pattern used to access data can impact performance. A sequential access pattern could run at near peak I/O throughput speeds. On the other hand, a random access pattern might access a different warp-line in memory for each individual thread within a thread warp, requiring 32 separate I/Os and wasting work (by loading 31 unused data elements per I/O request).

Coalescence: A single transfer operation (load, store) is fully coalesced, if a single data warp is accessed by a single thread warp in a single operation. Otherwise, if the data is spread out across multiple data warps, then the hardware must replay the transfer multiple times in order to access all the requested data.

Coalesced Access: See Coalescence.

Coalesced I/O Efficiency: A percentage measure of how efficiently *coalescence* is being taken advantage of. The percentage is the number of average data elements transferred per transfer divided by the maximum number of elements that could be transferred (k = 32). For example, if on average 16 out of 32 data elements are accessed on each store, then the Coalesced I/O efficiency is said to be 50% (=16/32). This efficiency metric has a minimum and maximum value of 3.125% (1/32) and 100% (32/32) respectively. Typically longer coherent data runs tend to require fewer transfer operations than shorter runs resulting in greater I/O throughput.

Coherence: see *data coherence, locality* or *coalescence*.

Coherency Cache Miss: An aliasing problem caused by virtual memory management where multiple virtual addresses all map onto the same physical address. A physical cache block may need to be evicted to make room for a new virtual address and then later retrieved as the program continues to execute.

Coherent Memory Access: Refers to memory accesses which support spatial locality, in other words, a group of memory accesses are typically within the same small neighborhood of memory as the first memory access. Sequential memory access (serial or parallel) is a perfect example of coherent Memory Access. Coherent memory accesses result in good *I/O throughput* as they support *caching* in modern memory architectures.

Coherent Masks: With SIMD warp branching, the situation where the branch masks across all processing cores in the warp contain either all zeros or all ones.

Collective Coordination: Multiple threads cooperate to communicate data and coordinate work with each other. Common coordination models include *broadcast model*, *scatter model*, *gather model*, and the *reduction model*.

Collective Operation: An operation, such as a reduction or a scan, that acts on the collection of data as a whole.

Collision: In the scatter pattern, or when using random writes from parallel tasks, a collision occurs when two or more parallel threads (or tasks) try to write to the same location at the same time. The result is typically *non-deterministic* since it depends on the timing of the parallel writes. In the worst case, unexpected garbage is written into the location impacting the correctness of the entire program.

Common Sub-expression Elimination: A *data-flow optimization technique* where a sub-expression that is computed two or more times in the original code but doesn't change values is only calculated once and then reused in the compiled code. See also *code fusion*, *constant folding*, and *strength reduction*.

Communication: Any exchange of data between parallel software tasks or threads. See also *coordination*.

Communication Avoiding Algorithm: An algorithm that avoids communication between tasks or threads, even it if results in additional or redundant computations.

Commutative Operation: A commutative operation $\oplus$ satisfies the equation $a \oplus b = b \oplus a$ for all a and b in its domain. Commutative operations allow data to be re-ordered but not re-grouped. Some techniques for parallelizing reductions and other operations require commutatively for correct results. See also *associative operation*.

Compare-and-Swap: Compare and swap (CAS) is an *atomic operation* used to synchronize *concurrency* issues and prevent incorrect multi-threaded behavior. It compares the contents of a memory location to a specific value, and if and only if they are the same, it then swaps the contents of the memory location with a given new value. Even though this actually requires several instructions in hardware, it appears as a single atomic operation to the rest of the threads, i.e. no other thread can interrupt the CAS once it has begun and until it has finished. Most modern CPU architectures implement one or more variations of Compare-and-Swap as atomic operations. CAS is often used to implement *lock-free* and/or *wait-free* algorithms and data structures. See also *fetch-and-add* and *test-and set* which CAS replaces.

Composability: The ability to use two components with each other. Can refer to system features, programming models, or software modules. If the two components can easily be made to work together than they are said to be composable, otherwise they are not.

Compulsory Cache Miss: The very first memory access to a memory block cannot be in the cache, so the cache block must be brought into the cache. Compulsory misses would occur even if the memory architecture contained an infinitely sized cache.

Complete: A complete binary tree is both a full binary tree and a left-balanced binary tree. In other words, all nodes (except leaf nodes) has two children; all levels except the last are completely filled, and on the last level all nodes are as far left as possible.

Concurrent: Logically happening simultaneously. Two or more threads (or tasks) that appear to all be logically active at some point in time are considered to be concurrent. Older *multi-threading* systems used pre-emptive multi-tasking on a single CPU to give the illusion of multiple tasks executing concurrently despite the reality that only a single instruction could execute at a time on the CPU. Contrast with *parallel*.

Concurrency Issues: The collection of issues associated with moving from *serial* to *parallel* execution. See also *non-deterministic*, *mutual exclusion*, *collision*, *deadlock*, *livelock*, and *starvation*.

Conflict Cache Miss: If the cache block placement policy is not fully associative, conflicts misses occur because multiple different memory blocks get mapped onto the same specific block in the cache and the current block in the cache needs to be evicted to make room for the currently requested block. The evicted block may need to be retrieved later as the program accesses it.

Constant Folding: A *data-flow optimization technique* that replaces a complex expression made up of only constant values with a single specific constant value that represents the entire complex expression. See also *strength reduction*.

Contention Issues: The collection of issues associated with multiple parallel threads (or cores) competing for access to shared resources. See also *non-deterministic*, *mutual exclusion*, *collision*, *deadlock*, *livelock*, and *starvation*.

Context Switch: With a context switch, A processing *core* p replaces one *thread* (or *warp*) s with another thread (or warp) t. s's state switches to *active* (or *blocked*, *inactive*) and t's state switches to *executing* (or *running*). This happens by pausing s on p, storing s's context, reloading t's context and re-starting execution of t on p. Context switches are done for two reasons 1) To keep processor cores busy by *latency hiding* and 2) To ensure each thread (or warp) gets a fair slice of processing time to make forward progress.

Control Dependency: A dependency between two tasks where whether or not a task executes depends on the result computed by another task.

Control Hazard: Control hazards, also known as branching hazards, are a direct result of branching on a pipelined architecture. Given that there are two distinct instruction paths {*true*, *false*} as a result of a branch, the instruction scheduler (hardware or software) doesn't know which set of instructions to start fetching & scheduling from until the branch condition outcome is actually known. *Stalling* is the original solution to this problem -- Wasteful NOPs are inserted into the instruction pipeline until the branch outcome is known and the program counter (PC) can be updated to start scheduling from the correct branch path. A *branch delay* is the number of stages (machine cycles) lost due to waiting on the outcome. *Branch prediction* and *branch speculation* are entire families of ILP optimization techniques intended to help reduce to number of stalls caused by control hazards.

Convergent Memory Access: When memory accesses by threads in a warp (or block) access adjacent memory locations. Actually the memory locations can be somewhat scattered as long as all accesses by an entire warp are constrained to a cache-line (warp-line) they can be in any order within the cache-line without causing a performance hit. See *divergent memory access*.

Cooperative Multi-tasking: With cooperative multi-tasking, also known as cooperative scheduling, the thread scheduling system allows threads to decide for themselves when to give up control of the CPU core to another thread. This approach is vulnerable to poorly written programs hogging most of the system resources for themselves leading to starvation for the rest of the better behaving applications.

Coordination: The synchronization of behavior between parallel software tasks or threads. See also *communication*.

Coordination Issues: The collection of issues associated with coordinating the behavior or communicating the exchange of data across multiple parallel threads (or cores). See *communication* and *coordination*.

Cooperative Thread Array (CTA): With a GPU running CUDA, a cooperative thread array (CTA) is the batch of threads launched by a GPU kernel to execute a parallel program. The CTA is organized as a *grid* of *thread blocks*. There are hardware constraints that limit the number of concurrent thread blocks that can execute on each individual SM core. But in aggregate it is potentially possible to run tens of thousands of threads concurrently on each GPU device.

Core: An individual processor within a multicore processor. Each core should be able to support at least one thread of execution.

Counting Sort: A *Counting Sort* sorts an unordered sequence *A* into an ordered sequence *S*. *A* contains *n* elements, where each element is represented by a numeric key in the range [0, *d*). (*d* is typically a small number, with $d \ll n$). These keys are used to distribute the *n* keys directly into *d* sorted *runs* using counting, indexing, and table lookup. The *d* runs are then concatenated to produce the final sorted output. By distributing keys using indexing and then concatenating runs, counting sort escapes the well-known $\Omega(n \cdot \log n)$ lower bound on comparison-based sorts. Counting Sort is often used as a subroutine within *Radix Sort*.

CPI: Cycles per instruction (CPI), the number of machine cycles it takes to execute a particular instruction on a core.

CPU: A central processing unit (CPU) is also known as a core, processor, or socket. Originally, a CPU was the single computation unit on a computer. Modern CPUs may actually contain multiple redundant processing cores which can be accessed in a pipelined or parallel manner. Multiple CPUs may be bundled together on a single motherboard to form a computer *node*.

CPU Time: The amount of time it takes a CPU to implement a program, task, method, or set of instructions. $CPU_{time} = Instruction_{count} \times Avg\ Clock\ Cycles\ per\ Instruction \times Clock\ Cycle\ Time$

Critical Path: The longest chain of tasks (or instructions) ordered by dependencies in a program.

CTA: See *cooperative thread array*.

CUDA: CUDA (formerly called the *Compute Unified Device Architecture*) is a high level language and parallel computing platform created at NVIDIA. The language is based on C++ and version 5.0 of CUDA uses the well-known Low Level Virtual Machine (LLVM) compiler infrastructure to compile, link, and generate run-time GPU code from CUDA kernel programs. The CUDA language includes several C++ extensions to express parallelism, data locality, memory usage, and to manage thousands of threads. The CUDA API supports many libraries for supporting kernel invocation, synchronization, memory allocation, memory transfers, numeric processing, random number generation, and exposing the rich parallel functionality available on each GPU device.

Cutting Plane: An axis-aligned hyperplane used to separate a set of *n* *d*-dimensional points (objects) from a parent node into two child nodes {*left*, *right*} as part of building a *spatial portioning data structure* such as a *kd-tree*.

Cycles per Instruction: See CPI

Cyclical: A cyclical *kd-tree* is one in which the *d*-dimensions are used as cutting planes in repeated cyclical order as often as necessary when building the *k*d-tree. For instance given 3D points with axes {*x*,*y*,*z*}. The cyclical cutting plane order would be {*x*,*y*,*z*,*x*,*y*,*z*, …} as often as required until the tree is fully built.

DASk: See *data access skeleton*.

Data Access Skeleton: A *data access skeleton* (DASk) is a machine independent data access pattern (or algorithm) adapted to run on a specific architecture like a GPU. DASks group *n* data elements into *m* fixed size data blocks. These *m* data blocks are then load balanced across *p* thread blocks according to a 1D or 2D layout. For instance, the Row DASk enables the parallel partition pattern on a GPU by supporting the top level of the 2-level CTA mapping while also preserving sequential ordering to support parallel patterns like reduce & scan. See also *block access skeleton*, *Block-*, *Column-*, and *Row-DASks*.

Data Alignment: When basic *data-types* or data structures are aligned to start on machine-word boundaries (32-bit or 64-bit), memory accesses tend to run much faster.

Data Block: A fixed-size unit of data elements that is meant to be processed in parallel by a matching thread block. With a *Data Access Skeleton*, an array of *n* data elements is grouped into *m* data blocks, each of size DBS and distributed across *p* thread blocks. The *data block size* (DBS) is typically parameterized using three template parameters -- *nWork, nWarps*, and *WarpSize*. These three parameters represent three different types of parallelism – *Instruction-level Parallelism*, *Thread-level Parallelism*, and *Data-level Parallelism*.

Data Block Size: A fixed-size *data block* which contains DBS elements, where DBS is typically computed as *DBS = nWork*nWarps*WarpSize*. See also *Thread Block Size*.

Data Coherence: A concept of how close data elements within a group are to each other. Data elements that are close to each other are said to have high coherence. Data elements that are far apart from each other are said to have low coherence. See also *locality*.

Data Compression: Data compression encodes information using fewer bits than the original representation. Compressing data can potentially improve performance by reducing processing instructions, transmission bandwidth and disk storage. However compressing and decompressing data can take a lot of computation which may slow-down performance.

Data Dependency: Two similar definitions 1) A dependency between two tasks where one task requires as its input data the output data from another prior task. 2) A dependency between multiple instructions where one instruction requires as its input operand(s) the output result(s) from prior instruction(s). See also *dependencies* and *data hazard.*

Data Dependent: See *data independent*.

Data-flow optimization: A family of optimization techniques (often done by a compiler) that seeks to improve performance by first analyzing then second reducing, replacing, or reordering instructions for better performance. See also *algebraic simplification*, *code fusion*, *common-subexpression elimination*, *constant folding*, *dead code elimination*, and *strength reduction*.

Data Hazard: Data hazards occur when multiple instructions in an instruction stream have dependencies on each other. Ignoring data hazards in *out-of-order* instruction *pipelined* architectures can result in race conditions which lead to incorrect program results. There are 3 main types of data hazards – *read after write (RAW)*, *write after read (WAR)*, and *write after write (WAW)* which are also known as *true-*, *anti-*, and *output-dependencies* respectively. Schedulers (hardware & software) must detect and handle data dependencies for correct program results. Typically this is done using *stalling* or *out-of-order* scheduling.

Data Independent: Given an algorithm with input data set of fixed-size *n*, if varying the data elements which makes up the input set has no noticeable impact on total performance, then the algorithm is said to be *data independent*. On the other hand, if varying the input data causes total performance to vary greatly than the algorithm is said to be *data dependent*. If varying the data set causes only minor performance fluctuations (say within 10% of the best), then the algorithm is said to be *partially data dependent*. For instance, in a histogram algorithm, replace a random data set by a set of all zeros, if the zero data set is 10× slower than the random data set than the algorithm is clearly data dependent.

Data-Level Parallelism (DLP): A type of *parallelism*. With data-level parallelism, a set of *n* data elements is partitioned (decomposed) into *m runs* that are then mapped onto *p* cores to execute in parallel. With a *equal balanced* partitioning, each core is responsible for processing *O*(*n*/*p*) elements as a 1D *run*, 2D *tile*, or 3D *block*. Data-level parallelism scales easily to handle more data or more parallel cores. See also *instruction-level parallelism* and *thread-level parallelism*.

Data Parallel Programming: With data parallel programming, programmers implement their parallel algorithms to take advantage of *data-level parallelism* on top of *SIMD* vector-parallel and/or *multi-threaded* hardware. The main idea is that each unique thread is assigned its own unique data element (or data run) to work on independently from every other thread concurrently working on its assigned data.

Data Structures: A data structure is a particular way of organizing data in a computer so that it can be used consistently and efficiently. Associated with each data structure is a set of procedures that create, delete, update, and manipulate specific instances of that data structure. Some typical examples of data structures include arrays, records, sets, lists, trees, graphs, and objects. See also *abstract data types* and *object-oriented programming*.

Data Throughput: The number of data elements processed per second during the execution of some program, algorithm, function, or section of code. Typically measured in mega-elements per second (ME/s) or giga-elements per second (GE/s).

Data Type: A data type is a classification identifying one of many specific types of data such as integer, floating point, Boolean, character, string, etc. The actual data type defines the values of that type; how that data should be interpreted; how many bits are required to represent that data type; and how that data should be stored in memory.

Data Warp: A collection of *WarpSize* (=32) data elements that are transformed from input into output by an algorithm implemented as one or more GPU kernels. Each data warp is associated with a matching *thread warp* and is executed by a set of multiple *SP* cores within each *SM* core. Data Warps naturally support *Data-level Parallelism* via *data parallel programming* to take advantage of *SIMD* hardware on a GPU. See also *Data Block*, *Thread Warp,* and *Thread Block*.

DBS: See *data block size*.

Dead Code Elimination: A *data-flow optimization* technique that detects and eliminates dead code, redundant code, or useless operations from the output code. For example, a complex calculation that is stored in a variable which is subsequently never used can be safely eliminated. See also *strength reduction*.

Deadlock: A *concurrency issue* that occurs when at least two tasks (or threads) circularly wait on each other to complete and each task (thread) will not resume execution until the other task (thread) proceeds. This error occurs frequently with code that uses *locks* for *mutual exclusion*. See also *livelock* and *starvation*.

Dependencies: A relationship among tasks, threads, or instructions that results in an ordering constraint. Dependencies are important to understand because they can inhibit parallel performance.

Depth: Depth, also known as span, is how long a program would take to execute on an idealized machine with an infinite number of parallel processors. The depth of an algorithm can be seen as the critical path in its task dependency graph.

Depth Complexity: Depth complexity, also known as *step*-, *circuit*-, and *span-complexity*, is an asymptotic measure of complexity in a parallel algorithm based on its depth. Informally, if we are willing to use more resources in parallel to solve a problem, we can often reduce the time required to solve that problem. Depth complexity, $D(n,p)$, measures how much time it takes for a given input size (n) and given number of parallel cores or threads(p). For the *work+depth analysis* of parallel algorithms, this measure is equally as important as *work complexity*.

Depth-First-Search: An algorithm for searching a tree or graph data structure. The algorithm starts at the root (or an arbitrary starting node) and explores as far as possible along each branch path before backtracking to other branch paths. Nodes to be visited are typically stored using a *Stack* in *LIFO* order. See also *Breadth-First-Search*.

Deque: A double-ended *queue*. Elements can inserted and removed from both the front and the back of the Deque.

Design Pattern: 1) A general term for pattern that includes not only algorithm strategy patterns but patterns related to overall code organization. 2) A general reusable solution to a commonly occurring problem in a given context. Design patterns are formalized best practices that the programmer must still implement themselves in a program. *Algorithms* and design patterns are orthogonal and can both be used at the same time.

Deterministic: A deterministic algorithm is an algorithm that behaves predictably. Given a specific input, a deterministic algorithm will always produce the same output no matter how many times it is run. Parallel floating-point algorithms may have small differences in output compared to the equivalent serial floating-point algorithms due to the different order in which the parallel algorithm groups floating-point data and operations compared to the serial algorithms and the fact that floating-point operations have small errors due to truncation and round-offs. However, in practice, concurrency is the main cause for non-deterministic results. Programmers need to use sequential semantics and eliminate all race conditions in order to convert non-deterministic algorithms into deterministic algorithms with the exception of non-determinism caused by “rounding” differences for floating point algorithms.

Device Processor: A device processor, also known as a *co-processor*, is a secondary computational unit that is under the control of a primary *host* processor. The device responds to requests from the host to do useful computations. *GPU*s are co-processor devices that can do massive parallel computation under the direction of a *CPU* host processor.

DFS: See Depth-First-Search.

Digit: A small numeric range specified by the *radix* (or base). For instance, a base 10 radix allows digits in the range [0,9]. See also *Key*.

Digit Value: A specific value taken from a digits numeric range. For instance, 5 is a specific digit value from the range [0-9].

Directed Acyclic Graph (DAG): A graph that defines a partial order so that nodes can be sorted into a linear sequence with references only going in one direction. A directed acyclic graph has, as its name suggests, only directed edges and no cycles.

Direct Memory Access (DMA): The ability of one processing unit to access another processing unit's memory with involving the other processing unit in the transfer.

Direct-Mapped cache: A cache organization in which data from multiple disjoint locations in main memory are mapped onto the same physical cache-line in the cache. Typically this direct mapping uses a modulo function of the original address. Older data needs to be evicted from cache anytime newer data is mapped onto the same cache line, which can lead to *cache thrashing* in some cases. Contrast with *associative cache*.

Distributed Memory: 2 Definitions: 1) Memory which is located in multiple physical locations. Accessing data from a remote location typically has higher *latency* and possibly lower *bandwidth* than accessing local memory. 2) Memory that is physically located in separate computers. An indirect interface such as *remote message passing*, is required to access memory on remote computers, while local memory can be accessed directly. Distributed memory is typically supported by clusters of computers.

Divergent Memory Access: When memory accesses by adjacent threads in a warp (or block) access scattered (non-adjacent) memory locations. See *convergent memory access*.

Divide and Conquer Pattern: Recursive decomposition of a problem. Can often be parallelized with the *fork-join* pattern.

Domain-Specific Language (DSL): A language with specialized features suitable for a specific application. For Example: MATLAB is a domain specific language for Matrix processing & Numerics.

DRAM: Dynamic Random Access Memory. See *RAM*.

Dynamic: A dynamic algorithm (or data structure) allows insertions, deletions, or other modifications to the original input data set (or data structure) as the algorithm proceeds. Contrast with *Static*.

Dynamic Scheduling: With dynamic scheduling, the thread scheduler (hardware or software) maps active threads onto available processing cores using pre-emption. The scheduler adapts to changing circumstances to try to optimize overall processor utilization by skipping stalled (or inactive) threads, waking up completed threads, blocking threads waiting on long running transfers of I/O resources, etc.

Dynamic Work Assignment: Dynamic work assignment is a technique for *load balancing* work across parallel threads (or cores). An array of *n* data elements is initially divided into *m* fixed-size chunks which are then assigned into a work queue. Each parallel core grabs a chunk of work from the queue and processes it until the work queue is empty. As new work is generated, it is also subdivided and added into the work queue. This approach works best when the total amount of work is variable or unpredictable. Preventing resource competition across threads when accessing or updating the work queue data structure can itself become a potential performance bottleneck. See also *equal partitioning* and *work stealing*.

Efficiency: Efficiency measures the return on investment in using additional hardware to operate in parallel.

Encapsulation: Three related definitions – 1) Encapsulation is the packing of data and functions into a single component. 2) A programming language mechanism for restricting access to some of the object's components, which is useful for *information hiding*. 3) A programming language mechanism for bundling data with the methods operating on that data. See also *object-oriented programming*.

Exceptions: Exceptions are the unusual situations that result from hardware instructions being presented with incorrect data that could result in incorrect system behavior if not handled properly. Hardware exceptions include overflow, divide by zero, translate look-ahead buffer miss for virtual memory, etc. Exceptions are usually handed off to special system software exception handlers to be dealt with properly. Handling exceptions often impacts program performance as the exception handlers often take hundreds or thousands of machine cycles to execute.

Elemental Functions: A function used in a *map pattern*. An elemental function acts on single item inputs. In other words each transformation from an input data element into an output data element has no dependencies on other data. An elemental function can be easily vectorized by replicating the computation it specifies across multiple SIMD lanes. See *embarrassingly parallel algorithms*.

Embarrassingly Parallel Algorithms: With *embarrassingly parallel algorithms*, the *depth efficiency* is constant, $D(n) \approx O(1)$, in other words the number of parallel steps required is fixed, small, and independent of the input size. These algorithms process each data element completely independently from all other data elements so no communication or coordination across cores is required. The actual code tends to follow the *map pattern* and is often trivial.

Equal Partitioning: Is a technique for doing *load balancing* across parallel cores. Given *n* data elements or operations. Each of the *p* parallel cores is assigned to process equal sized chunks $\lceil n/p \rceil$ of work (elements or operations). This approach works best when the amount of work is known *a priori* (in advance) and the data or operations are uniformly sized. See also *dynamic work assignment* and *work stealing*.

Exclusive Scan: A type of *scan* where the *i*th output element (s_i) is the total sum of the first (i-1) elements from an input array *A* under some binary associative operator $\oplus$ with identity $\mathbb{I}$, in other words $s_i = \mathbb{I} + \sum_{j=1}^{i-1} a_j$. Compare with *Inclusive Scan*.

Expand Pattern: A pattern in which each element of a *map pattern* can output zero or more data elements, which are then assembled into a single (possibly segmented) array. Compare to the *pack pattern*.

False Sharing: Two separate tasks in two separate cores may write to separate locations in memory, but if those memory locations happen to be allocated into the same cache line, the *cache coherence* hardware will attempt to keep the cache lines coherent, resulting in extra communication and reduced performance, even though the tasks are not actually sharing data.

Faster: An improved method *S* is said to be *x* times faster than a baseline method *T*, which is expressed using the following formula.

$$x = \frac{Execution\ Time_{old}}{Execution\ Time_{new}} = \frac{Performance_T}{Performance_S}$$

Fetch-and-Add: fetch and add is an *atomic operation* used to synchronize *concurrency* issues and prevent incorrect multi-threaded behavior. It adds a new value to a value in a memory location and then returns the old value from the same memory location. Even though this actually requires several instructions in hardware, it appears as a single atomic operation to the rest of the system, i.e. no other thread can interrupt the CAS once it has begun and until it has finished. This atomic operation has been replaced by *compare-and-swap* semantics on modern architectures. See also *test-and-set*.

Fiber: A very lightweight unit of execution. Similar to a thread, but fibers use co-operative multi-tasking while threads use pre-emptive multi-tasking.

Fibonacci Numbers: The Fibonacci numbers are defined by a linear recurrence and are often used as examples of recursion in computer science. The two base cases are defined as $F(0) = 0$ and $F(1) = 1$ with the recursive induction step, $F(N) = F(N\text{-}1) + F(N\text{-}2)$. The integer sequence grows as $\{0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots\}$.

FIFO: Abbreviation for *first-in, first-out*. For instance a single line of customers in a book store are typically handled in FIFO order. The first customer to enter the service *queue* is the first customer to reach a cash register and check out. Contrast with *LIFO*.

Fill: A type of *map* primitive, which fills a destination array *D* of *n* elements with a single data value. For example: setting an entire array to all zeros is a form of fill. Also see *Copy*, *Scatter*, and *Gather*.

Fine-Grained Locking: *Locks* used to protect small parts of a larger data structure from race conditions. Such locks avoid locking the entirety of the large data structure during parallel accesses.

Fine-Grained Parallelism: A level of parallelism with very small units of parallel work per unit of coordination (communication). Care must be taken to reduce the cost of coordination (communication) otherwise *parallel overhead* will dominate performance costs and reduce parallel efficiency.

Fine-Grained Multi-threading: With fined-grained multi-threading, the scheduler can and does switch between active threads on a core every few cycles (even as often as once per cycle). Thread contexts must be *light-weight*. The thread scheduler can adapt to short-term stalls in execution by pausing threads (warps) and switching to other active threads.

Floating-Point Data: Modern GPUs support both 32-bit and 64-bit floating point data types to approximate real numbers and associated arithmetic operations as specified in the IEEE 754 specification.

Floating-Point Unit: See *FPU*.

Floor Function: Given any real number x, there exists a unique integer n such that $n \leq x < n+1$. We say that n is the floor of x, often denoted as $\lfloor x \rfloor = n$. See also *ceiling function*.

Flynn's Taxonomy: A well-known categorization of parallel processing into four groups {*SISD*, *SIMD*, *MISD*, and *MIMD*} based on parallel instruction handling {*single*, *multiple*} and parallel data handling {*single*, *multiple*}.

Fold: An operation on a collection of data in which every output is a function of all previous outputs and all inputs in a sequence up to the current index. A fold is based on a *successor function* that computes a new output value and new state for the fold from the previous state and each new input value. A *scan* is a special parallelizable case of a fold where the successor function is associative.

Fork: The creation of a new thread or task from an original thread. The original thread or task continues executing in parallel with the forked thread or task. See also *spawn* or *join*.

Fork-Join Pattern: A parallel pattern of computation in which new (potential) parallel flows of execution are created/split using *forks* and terminated/merged with *joins*.

Fork Point: A point in the code where a *fork* takes place.

FPU: An FPU (floating-point unit) is a small processing unit which can perform simple computations such as addition or multiplication on floating-point data. See also *ALU*.

Frequency Scaling: A way to increase computational throughput (performance) by increasing the frequency at which transistors are run. As transistors get smaller, it gets easier to run them faster. However, faster running chips also generate more heat. If chips get too hot they will burn out. So frequency scaling requires careful cooling. Modern chips can't run much faster than ~4 GHz on air-flow cooling (fans) without risk of burn out. More exotic forms of cooling like water, oil, liquid nitrogen, etc. can be used to raise this maximum frequency.

Frequency Scaling Era: The time period from roughly 1965-2004 where the primary way to increase integrated chip performance was to increase the frequency at which transistors ran. In 2004, chip architectures hit a *power wall* where running chips faster than ~4 GHz generated more heat than air-flow cooling (fans) were capable of removing from the chip risking burn out. In 2005 chip designs moved into the *Multi-core era*.

Full: A Full binary tree is a binary tree in which every node other than leaf nodes has two children.

Fully Loaded: A fully loaded memory access pattern uses all the data in each cache-line before accessing data that would result in another cache line being loaded into cache. A sequential access pattern over a large array results in one I/O per cache line and all the data in each cache line being used fully before moving onto the next cache line. This is true for all cache lines, except for possibly the first and last cache lines which might only be partially full with data due to alignment issues. Whereas, a random access pattern loads an entire cache line but only accesses a single data element. That cache line is highly likely to be evicted from cache before being randomly accessed again resulting in lots of I/O and poor utilization of the caching architecture. See also *coherence*, *coalescence* and *alignment*.

Functional Decomposition: *Task-level parallelism* where the original task is divided into smaller sub-tasks and each sub-task is then run on its own thread. A *directed acyclic graph (DAG)* is often used to model the flow of data through the sub-tasks. This approach is difficult to *scale* beyond a small number of threads.

Functional Unit: A hardware processing unit that can perform simple computations such as addition or multiplication. See also *ALU* and *FPU*.

Function: A function typically consists of a function-call interface used to invoke the function from elsewhere in the program and the body of the function which is the actual sequence of instructions that perform some specific task for which the function was written. In C++, the function-call interface is called a declaration and the function body is called the definition. Each function has one or more input and output *parameters*. Functions enable code reuse and modular programming.

Function Overloading: Function overloading (or method overloading) is a type of *polymorphism* that creates multiple methods with the same name but with possibly different implementations for different data types. For instance: One could create a swap function for exchanging two integers, and another for swapping two strings. The integer swap could take constant time $O(1)$, while the string swap could require linear time $O(n)$.

Functor: A class which supports a function-call interface. They behave similarly to C or C++ callback functions and can be used in the same manner. However, functors are also objects and can hold state and support additional class interfaces for modifying that state. Functors can often be optimized by compilers (typically removing the function invocation overhead) often resulting in better performance than an equivalent callback function.

Fusion: An optimization in which two or more things with similar forms are combined to reduce overhead. See *loop fusion*, *cache fusion*, and *code fusion*.

Gather: A type of *map* primitive, which gathers elements from a large source array S into a small destination array D using an index map. Also see *Copy*, *Fill*, and *Scatter*.

Gather Model: A *collective coordination* model, where all threads (but one) send a message to a single primary thread. The primary thread then composes the messages together.

Gather Pattern: A set of parallel random reads from memory. A gather takes a collection of memory addresses and outputs a collection of data read from those memory addresses. Gathers are equivalent to random reads inside a *map pattern*. Given an input array of size n, and an output array of size m, and an index array of size *m*, typically with $n \geq m$. Each element of the output array comes from some element in the input array at the given index. *Output*[*i*] = *Input*[*index*[*i*]] for all $i \in [0, m)$. The *scatter pattern* is the inverse of the gather pattern.

Generic Programming: A style of programming in which algorithms can be written in terms of generic abstract types which are then instantiated later as the specific data types needed by a specific implementation. C++ *templates* are a mechanism for supporting generic programming. *Data access skeletons* use generic programming techniques to enable performance experiments with three template parameters – *nWork, nWarps,* and *WarpSize* representing three types of parallelism – *instruction-level parallelism, thread-level parallelism,* and *data-level parallelism*.

Geometric Decomposition Pattern: A pattern that decomposes a data domain for an algorithm into a set of possibly overlapping sub-domains. A special case is the *partition pattern*, which is when the sub-domains do not overlap.

GPU: A GPU (graphics processing unit) was originally developed to accelerate graphics computations but now is sufficiently evolved to support arbitrary computations. GPUs are especially good at massively parallel, fine-grained data-parallel algorithms. They typically use *multi-threading*, *hyper-threading*, SIMD threads, and extensive use of *latency hiding*.

GPU Kernel: A GPU kernel is a program that runs in parallel on a GPU device. A GPU kernel looks like a serial program written to run on a single thread. However, the GPU device will run the kernel on many threads in parallel.

GPU Programming Model: A programming model where parallel computation is run on a GPU co-processor device using a *CTA* (a 2-level *grid* of thread *blocks*). The parallel computation is done under the direction of a CPU host core. This model supports limited communication and synchronization between different threads within the same thread block. But assumes no communication or synchronization between different thread blocks in the same grid. This model makes no assumptions about what order thread blocks in a grid are scheduled onto the SM (SMX's) processing cores by the GPU. A coarse level of synchronization is supported via *kernel termination* in that all thread blocks in a grid are guaranteed to have completed when the kernel terminates.

Grain: A unit of work to be run serially on a single core. See *granularity*.

Grain Size: The amount of work in a *grain*.

Granularity: Granularity is a qualitative measure of the ratio of computation to *coordination* (communication) in a parallel algorithm. The amount of decomposition applied to the parallelization of an algorithm, and the grain size of that decomposition. *Coarse granularity* implies large amounts of computations are done between communication events. *Fine granularity* implies that small amounts of computations are done between communication events. If the granularity is too coarse, then there won't be enough parallel threads (or tasks) to keep all the parallel cores busy (*under-utilization*) or poor *latency hiding*. If the granularity is too fine, then the coordination required between parallel threads may dominate the resulting performance. See also *parallel overhead*.

Graphics Accelerator: A co-processor specialized for handling graphics workloads, typically used for real-time graphics. APIs such as Direct3D and OpenGL are typically used to send graphic commands to the hardware. See also *GPU*.

Graph Rewriting: A computational pattern where nodes of a graph are matched against templates and substitutions made with other sub-graphs. When applied to directed acyclic graphs (DAGs), this is known as term graph rewriting and is equivalent to the lambda calculus, except that it also explicitly represents memory sharing.

Greedy Scheduling: A scheduling strategy in which no worker idles if there is work to be done.

Grid: Two different definitions: 1) A distributed set of computers that can be allocated dynamically and accessed remotely. A Grid is distinguished from a cloud in that a grid may be supported by multiple organizations and is usually more heterogeneous and physically distributed. 2) A batch of *thread blocks* meant to be *concurrently* executed on a GPU to carry out a *GPU kernel* program. Typically a grid is 1D or 2D in shape. See also *CTA*.

Gustafson's Law: A different view from Amdahl's Law that factors in the fact that as problem sizes grow the serial portion of computations tend to shrink as an overall percentage of the total work to be done. Given α and P, where $\alpha \in [0,1]$ is the percentage of the program which is serial and P is the number of processing cores used to parallelize a program. Then the speedup according to Gustafson's Law is $S(\alpha, P) = P - \alpha \cdot (P - 1)$. Compare and Contrast with *Amdahl's law* and *Work+Depth analysis*.

Halo: In the implementation of the *stencil pattern* on *distributed memory*, each thread often needs to access nearby neighbors outside of its initial assigned data partition. The halo is the set of boundary elements surrounding each partition that are replicated on different worker threads to allow each partition to be computed in parallel.

Hash Lookup: A hash function is a well-defined procedure or mathematical function that converts a large amount of data (variable or fixed) into a small datum. This datum can be turn be used as an index to speedup searching, sorting, or table-lookups. If the resulting datum is small enough, a "trivial hash function" can be used to provide constant time searches at almost zero cost. See also *indexing*, *table-lookup*.

Hazard: A hazard is any parallel computing approach which could end up producing incorrect results compared to an equivalent serial approach. The main types of hazards include *structural hazards*, *control hazards*, and *data hazards*.

Heap: Two definitions 1) An abstract data structure for loosely ordering data in a conceptual tree, this data structure can be used for priority queues and sorting. 2) The physical memory used for storing allocation requests made using heap allocation, this is also known as the free store or dynamic memory.

Heap Allocation: An allocation mechanism, also known as *free storage* or *dynamic memory allocation*, that supports unstructured memory allocations of different sizes and at arbitrary times during a program's execution. Compare with *stack allocation*.

Heavy-weight Context: A heavy-weight thread (or warp) context typically contains 1K or more of execution state can take hundreds of machine cycles or more to transfer into/out-of virtual memory during a context switch. Heavy-weight contexts implies the use of *coarse-grained parallelism* due to the large overhead of heavy-weight context switches.

Heterogeneous Computer: A computer which supports multiple processing cores, with each core having its own unique specialized capabilities and/or performance characteristics.

High Performance Computing: High performance computing (HPC), also known as super-computing, uses the world's fastest and largest computers to solve massive and complex problems. Modern supercomputers typically use thousands of GPUs and CPUs in parallel.

Histogram: A histogram summarizes the frequency distribution of an entire data set via a much smaller table of counts. In a nutshell, *n* input elements are counted into *m* bins. The resulting *m* frequency counters form the histogram. A bar chart is often used to visualize the histogram. All *n* inputs are assumed to be taken from a range $R = [min, max)$. Each of the *m* bins represents a sub-range r_i from *R*. (These sub-ranges uniquely partition and fully cover the original range *R*). Counting proceeds by selecting the sub-range that contains each input element and incrementing the corresponding bin's counter.

Host Pattern: A simple programming pattern for launching *GPU kernel*(s) from a CPU host program. The pattern involves 1) allocating GPU & CPU resources 2) transferring data from the CPU onto the GPU 3) launching the GPU kernel(s) 4) transferring results from the GPU to the CPU and 5) deallocating resources.

Host Processor: 2 different definitions: 1) The main control processor in a system as opposed to any co-processors or GPUs also in the system. The host processor is responsible for booting and running the operating system. 2) The CPU processor that coordinates running a parallel algorithm using one or more GPUs.

HPC: See *High performance computing*.

Hybrid Parallelism: A type of parallelism that combines two or more other types of parallelism to support parallel processing in a parallel system.

Hyperplane: An arbitrarily oriented *d-1* dimensional plane used to separate a set of *n d*-dimensional points (objects) from a parent node into two child nodes {*left*, *right*} as part of building a *spatial portioning data structure* such as a *BSP tree*. See also *cutting plane*.

Hyper-threading: Multi-threading on a single processor core. With hyper-threading, also known as *simultaneous multi-threading (SMT)*, one core draws instructions from multiple threads to fill multi-issue instruction pipelines.

Identity: An identity element, $\mathbb{I}$, is a data element under some binary operator $\oplus$ such that $a = a \oplus \mathbb{I} = \mathbb{I} \oplus a$ for all $a \in \mathbb{U}$. For instance, one is an identity element for multiplication, any number times one returns itself. See also *Reduce* and *Scan*.

Idle Cycle: any machine cycle where a processing core is not doing useful work. When a program is running, there are multiple causes (*ramp-up/ramp-down time*, *hazards*, *cancellations* …) which lead to idle cycles. See also *stalls*.

ILP: See *instruction-level parallelism*.

ILP Wall: The limits to automatic parallelism given by the amount of parallelism naturally available at the instruction level in serial programs. This wall was effectively reached in the early 2000's when new ILP features that gave at best a 1-3% performance gain over existing ILP approaches cost significant amounts of silicon real estate to implement in hardware.

Information Hiding: Information hiding segregates the design decisions about the interface for using an algorithm from the actual implementation of that algorithm. The goal is to provide a stable interface (which seldom changes) for other programmers to use a specific function, class, module, or program while hiding the implementation details from those programmers (this allows the underlying implementation to change more frequently as needed to fix bugs and improve performance). In other words, the details of the implementation becomes a black box from the point of view of the user, all that matters is the interface which allows access to the desired behavior. *Encapsulation* is often used to enforce information hiding and these two terms are often used interchangeably. See also *object-oriented programming*.

Implementation: A specific section of code, function, module, or program that realizes a software design to solve a problem. The implementation is written by a programmer in a specific programming language and then compiled and linked into an executable program. The program solves the problem when given correct inputs and run. The implementation may use various algorithms, design patterns, and idioms in solving the problem. Some implementations are better than others. A good implementation should be correct, robust, and fast. Correct means the code solves the original problem as intended. Robust means that the code doesn't crash even when given incorrect inputs. Fast means the code runs with sufficient performance that an end user is willing to wait for the output results (the code return results in seconds, minutes, or hours not after months, years or decades).

Implementation Pattern: A pattern that is specific to efficient implementation using specific hardware mechanisms.

Incoherent Memory Access: Refers to memory accesses which don't support (or only weakly support) spatial locality. In other words, a batch of memory accesses will tend to be scattered across the memory address space. Each memory access will tend to bring in an entire cache-line into cache just to read one data item (or byte). Randomly picking data from a data set is a good example of incoherent memory access. Incoherent memory access results in poor *I/O throughput* as it under-utilizes *caching* which is based on *spatial locality*.

Inclusive Scan: A type of *scan* where the i^{th} output element (s_i) is the total sum of the first i elements from an input array A under some binary associative operator $\oplus$ with identity $\mathbb{I}$, in other words $s_i = \mathbb{I} + \sum_{j=1}^{i} a_j$. Compare with *Exclusive Scan*.

Indexing: Is an optimization technique that represents a large data object by a simple index often called a *key* to improve the performance of searches, sorts, and table lookup. Indexing usually assumes a trivial function for computing the index from the original data object. Often the index (or key) is stored as part of the original data object. See *hash look-up*.

Induction: When proving a statement true, the *principle of mathematical induction* is often used. For proving computer science algorithms correct based on induction, a third requirement of *Termination* is added to the *Basis* and *Inductive Steps*. The Termination step needs to prove that the algorithm will actually end after a finite number (n) of steps. To implement an algorithm based on induction, the *iteration* pattern is often used.

Inheritance: In object-oriented programming, inheritance is when one object or class called the child class is based on (derived from) another object or class called the parent class and inherits the same default behavior (interface and methods) from the parent class. However, certain "virtual" methods may be specified as *polymorphic* – there is a default implementation in the parent class and a possibly different (overloaded) implementation in the child sub-class. For instance, a draw method in a parent shape class may do nothing but the overloaded draw method in a child "box" class might know how to draw a rectangle based on the specific attributes of that object (width, height, color). See also *object-oriented programming*.

Inlining: Inlining, also known as inline expansion, is an optimization technique implemented by a programmer or compiler to replace a relatively short function call with the entire function body instead. This optimization often may improve runtime performance by reducing function call overhead and giving optimizing compilers more opportunities for optimizations without being stopped by function boundaries. On the negative side, the resulting application binaries are often larger and the larger instruction sizes may result in more instruction cache misses which slow down performance.

In-Order Execution: With in-order execution, the hardware executes a program (instruction stream) sequentially in the exact same order that the instructions were laid out in the original program. Contrast with *out-of-order execution*.

Instance: In a *map pattern* one invocation of the *elemental function* on one data element of the map.

Instruction-Level Parallelism (ILP): A type of *parallelism*. Instruction-level parallelism (ILP) exploits parallelism found in a sequential instruction stream by re-ordering, grouping, and executing independent instructions in parallel without changing the actual program results (as compared to a simple serial computer).

Interleaved Scheduling: With interleaved scheduling, the thread scheduler (hardware or software) switches between threads every few cycles (as often as once per cycle). All active threads (or warps) are executed in turn, typically in a round-robin fashion. As an optimization, *stalled* (or *blocked*) threads are skipped over. This schedule is inherently fair but may result in under-utilization of the processing core.

Intrinsics: Special function wrappers which allow direct access to a hardware's specific instructions. They appear to programmers as normal functions but are replaced by the compiler by a short sequence of assembly instructions that map directly into the desired hardware instructions. See also *instruction-set architecture (ISA)*.

Instruction-Set Architecture (ISA): An Instruction Set Architecture (ISA) is a set of operations (arithmetic, logic, control, data transfer, etc.) exposed by the hardware for performing computations. Programmers commonly access the ISA indirectly via a compiler which transforms instructions from a high-level language (such as C++) into machine language instructions for a specific hardware architecture (such as PTX for NVIDIA GPUs). The ISA can also be accessed directly by a programmer using *intrinsics*.

Instruction Throughput: The average number of instructions processed per second over the duration of a program, algorithm, or function. Typically measured in mega-instructions per second (Mi/S) or giga-instructions per second (Gi/S).

Instructions per Cycle (IPC): Instructions per cycle (IPC) is the average number of instructions retired per cycle on a particular computer hardware architecture. See also *multi-issue.*

I/O Throughput: The number of bytes transferred per second during the execution of a program, algorithm, or function. Typically measured in mega-bytes per second (MB/s) or giga-bytes per second (GB/s).

Irregular Parallelism: Parallelism using heterogeneous tasks or different sized data runs across threads. Irregular work makes load balancing work across threads more difficult.

Iteration Pattern: A serial pattern where the same set of instructions is executed repeatedly and in sequence.

Join: A join is where multiple threads of execution are merged into a single thread of execution which continues on from the *join point*. All other threads (in the join) are terminated or re-used for other purposes. See also *fork*.

Join Point: The point in the code where a *join* takes place.

***k*d-Tree:** A *k*d-tree is a hierarchical *spatial partitioning data structure* that is used to organize points (or other geometric objects) in *d*-dimensional space for faster geometric searches. A *k*d-tree can be built in $O(n \cdot \log n)$ time, takes $O(d \cdot n)$ storage, and can find the closest point to a query point in $O(d \cdot n^{(1-1/d)})$ worst-case time, with an expected time of $O(d \cdot \log n)$. See also *BSP Tree*, *octree*, and *nearest neighbor search*.

***k*NN:** See *k-Nearest Neighbor*.

***k*-Nearest Neighbor:** A type of *Nearest Neighbor Search* (*k*NN) where each query point in a query set *Q* is matched to the *k* closest points in a search set *S* under some distance metric. See also *k*d-tree.

Kernel: 3 definitions: 1) Key code sequence for an algorithm 2) Small section of code that performs the crucial portion of an algorithm. 3) On GPUs, a Kernel is the official name for functions launched by the CPU host but executed on the GPU device to implement a *data-parallel* algorithm.

Kernel Termination: Kernel termination is waiting until a GPU kernel has fully completed execution before launching the next GPU kernel as part of a larger parallel algorithm. This allows a very coarse-grained form of parallel *coordination* or *barrier synchronization* across all the threads in a *CTA*.

Key: For comparison-based sorts, a key is the primary value used to compare two objects to determine their relative sorted order (less than, equal, or greater than). For *Radix Sort*, A key is a numeric value taken from a large fixed-size numeric range [0, *m*). A key is written in a base, or *radix,* as a string of *digits*, where each specific digit value is taken from a small fixed-size numeric range [0, *d*). Typically, the radix is much smaller than the keys range, $d \ll m$. For Example: A 32-bit number using a base 16 radix would require at most a string of 8 hexadecimal digits (=$\lceil \log_{16} 2^{32} \rceil$) to represent.

Lambda Expression: An expression that returns a *lambda function*.

Lambda Function: A lambda function is an anonymous function. This feature has been part of languages like LISP for a long time but was only recently added to C++ per the C++11 standard. A lambda function enables a fragment of code to be treated as a function parameter and passed into another function without having to write a separately named *function* or *functor* for this purpose.

Lane: A lane is the idea of focusing on a single item within some larger group of which it is a part, similar to a lane within a road. Three similar definitions: 1) A single SP core within a SM(X) core is sometimes called a lane. 2) A single *thread* within a *thread warp* or *thread block* is sometimes called a lane. 3) A single data element(s) being worked on by a single thread within a thread warp or block is often called a lane.

Latency: The duration of time it takes to issue a request until its corresponding response is received. Latency is measured using units of time such as machine cycles, nano-seconds, seconds, days, etc. Latency *stalls* on computers can be long (hundreds of machine cycles) often caused by waiting on I/O operations or other system resources, or short (20 cycles or less) often caused by hardware schedulers issuing NO-OP commands to overcome *hazards* that would otherwise jeopardize correct behavior. CPUs are sometimes described as latency-focused architectures as the goal is to minimize the total running time of a single application.

Latency Hiding: On older architectures, when a thread of execution would *stall* waiting on some response, the core would often idle doing no useful work. Modern GPU hardware schedulers can avoid unnecessary idling by 1) switching from the stalled thread warp to another active warp (aka *Thread-Level Parallelism* TLP; or by 2) finding other independent instructions that can be executed now safely (aka *Instruction-Level Parallelism* – ILP). The process of keeping cores busy in the presence of latency stalls via TLP (other warps) or ILP (other instructions) is known as *latency hiding*. Latency hiding increases processing *throughput*.

Left-Balanced: A balanced binary tree in which the left sub-tree of each node is filled before the right sub-tree is filled. Informally, leaf nodes only occupy the leftmost positions in the last level of the tree.

Left-Balanced Median Position: Given *n* points in a parent node, the left-balanced median position (LBMpos) is the point which would split the parent node into two child nodes {left, right} which eventually would form an entire left-balanced tree. The LBMpos can be found in 3 steps as follows: 1) $h = \lceil \log_2(n+1) \rceil$ 2) $half = 2^{(h-2)}$ 3) $LBMpos = half + \min(half, n - 2 * half + 1)$. Note: The above formulas do not work correctly for small values ($n \leq 3$). The correct LBMPos for n = {1, 2, 3} are {1, 2, 2} respectively.

LIFO: Abbreviation for *last-in*, *first-out*. For instance a group of people using an elevator typically do so in LIFO order. The first people to enter the elevator move to the back to make room for others and are typically the last ones off the elevator when reaching the destination floor. Contrast with *FIFO*.

Light-weight context: A light-weight thread (or warp) context contains just a few items (typically 32 registers or less) and can be context switched quickly by a hardware scheduler (in just a few machine cycles). Light-weight contexts support *fine-grained parallelism*.

Little 'o' Notation: Complexity notation that denotes an asymptotic strict upper bound on resource usage; written as $o(g(n))$. Little 'o' notation is useful for comparing the algorithmic efficiency of different algorithms. $o(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) < c \cdot g(n)$ for all $n \geq n_0$ $\}$. See also *asymptotic complexity*, *big 'O' notation*, *big 'omega' notation*, *little 'omega' notation*, and *big 'theta' notation*.

Little 'omega' Notation: Complexity notation that denotes an asymptotic strict lower bound on resource usage; written as $\omega(g(n))$. Little 'omega' notation is useful for comparing the algorithmic efficiency of different algorithms. $\omega(g(n))= \{f(n)$: there exist positive constants c and n_0 such that $0 \leq c \cdot g(n) < f(n)$ for all $n \geq n_0$ $\}$. See also *asymptotic complexity, big 'O' notation*, *Little 'o' notation*, *big 'omega' notation*, and *big 'theta' notation*.

Linear Speedup: Speedup where performance improves in direct proportion to the computation resources used to solve the problem. Achieving a linear speedup is considered optimal for parallel programming.

Little's Law: A formula relating parallelism, concurrency, and latency. This theorem by John Little comes originally from queuing theory. Reinterpreted for parallel processing, in a parallel system that has reached steady-state equilibrium, it ties the average request rate (λ), the average time each request takes to be processed (W), and the number of concurrent requests pending in the system (L) as follows: $L = \lambda W$.

Live-Lock: A *concurrency issue* where multiple threads (or tasks) are *active* yet cooperate to block each other from making forward progress towards work completion. Similar to two people trying to pass each other in a hallway but they both keep stepping into each other's way. See also *deadlock* and *starvation*.

Load Balancing: Refers to the practice of distributing work among multiple threads (or tasks) so that all threads (or tasks) are kept busy most of the time. Load balancing can be done using *equal partitioning*, *dynamic work assignment*, and *work stealing*.

Load Imbalance: A situation where distributing uneven sizes of *tasks* or *runs* onto worker threads results in some threads finishing early and then idling while waiting for other threads to complete the larger tasks or runs.

Locality: The notion that a data value in storage (or nearby values) will be frequently accessed during the course of an executing program. Systems with strong locality are good candidates for optimizations which exploit locality, such as caching, pre-fetching, pipelining, branch prediction, etc. There are two main types of locality – *temporal locality* and *spatial locality*.

Local memory: Local memory is a CUDA technique where *register spills* (extra variables) are put into global memory and accessed at global memory speeds. This enables kernel compilation for complex algorithms at the cost of relatively slow I/O for the affected variables.

Lock: A generic term for objects used to ensure *mutual exclusion* between multiple parallel *thread*s (or *task*s). Only one thread (or task) at a time may own the lock. Other threads (or task) may attempt to acquire the lock but must wait until the owning thread (or task) releases the lock.

Lock-Free Algorithm: A *non-blocking algorithm* is lock-free if there is guaranteed system-wide progress across all threads. See also *wait-free algorithm*.

Loop: A sequence of code which implements the *iteration pattern*. Common loops include **`while`** `(test) {…} statements`, **`do {…} while`** `(test)` statements, and **`for`** `(setup; test; increment) {…}` statements.

Loop-carried dependency: A dependency that exists between multiple iterations of a sequence of code implementing the iteration pattern.

Loop Fission: A *loop optimization* technique where a single loop with a complex body is broken into two or more loops with simple bodies. The goal with this technique is to improve *locality* when accessing I/O arrays for better performance. Contrast with *loop fusion*.

Loop Fusion: A *loop optimization* technique where two or more loops with compatible indexing executed in sequence are rewritten and merged into a single loop. This technique can help reduce loop overhead and the total I/Os into input & output arrays. Contrast with *loop fission*.

Loop Interchange: A *loop optimization* technique where a compiler or a programmer exchanges the order of a set of nested loops (inner-outer ↔ outer-inner) for better I/O performance. Typically the change is made to better align the data access pattern into memory with the underlying cached memory architecture layout.

Loop Invariant Code Motion: If a value is computed inside a loop during every loop iteration but yet has the same value on each loop iteration. Performance can be improved greatly by moving the value's calculation outside the loop, where it is computed just once, stored in a register, and then reused as needed inside the loop.

Loop Optimizations: Loop optimizations are a family of implementation optimization techniques targeted at increasing program performance by reducing the overhead of loops. Since loops are frequently used and executed repeatedly, they are good targets for optimization. Loop optimizations include – *loop fission*, *loop fusion*, *loop interchange*, *loop invariant code motion*, *loop reversal*, *loop unrolling*, and *loop unswitching*. See also the related concepts *software pipelining* and *tiling*.

Loop Overhead: The necessary instructions (increments, branch tests, etc.) needed to iterate repeated over a loop which implements the iteration pattern.

Loop Reversal: A subtle *loop optimization* technique which reverses the order in which values are assigned to the index variable. This technique can sometimes eliminate loop carried dependencies and enable other loop optimizations.

Loop Unrolling: A *loop optimization* technique performed manually by a programmer or automatically by a compiler to improve performance by reducing loop overhead. With loop unrolling, instead of processing one array element per loop iteration, we process k array elements per iteration, this reduces pointer arithmetic and amortizes the cost of loop overhead across k elements. This increases register usage from $O(1)$ to $O(k)$. Care must be taken to deal with the last few left-over data elements [1..k-1] which don't fit into a single loop iteration. See also *software pipelining*.

Loop Unswitching: A *loop optimization* technique where a conditional inside a loop is moved outside of the loop by duplicating and rewriting the loop twice once each for the {*true*} and {*false*} clauses.

Many-Core: A processor with tens, hundreds, or even thousands of parallel processing cores.

Many-to-One: A mapping relationship where many source objects are related to one destination object in a set, table, or array of parallel objects. See also *cardinality*, *one-to-one, one-to-many*, and *many-to-many*.

Many-to-Many: A mapping relationship where many source objects are related to many destination objects in a set, table, or array of parallel objects. See also *cardinality*, *one-to-one*, *one-to-many*, and *many-to-one*.

Map Pattern: With a map pattern, a single *elemental* function is applied to all the data elements in an array. The data elements are mapped in parallel onto multiple processing threads (or cores). The *elemental function* requires no coordination between parallel threads resulting in *embarrassingly parallel* performance.

Masking: Two definitions: 1) Bit-level technique for retrieving only the desired bits from a larger machine word of bits. 2) Another term for how GPUs implement vectorized branching across a *warp* of threads using *serialization* and *predication*. See also *Branch Divergence*.

Massively Parallel: Refers to hardware that comprises a parallel system having many processing cores. The meaning of "many" keeps increasing, but currently, means hundreds, thousands, or even hundreds of thousands.

Mathematical Induction: See *Principle of Mathematical Induction*.

Median of Three: *Quicksort* tends to slow down to quadratic time $O(n^2)$ when encountering sorted (reverse sorted) or nearly sorted input data. The problem is that each pivot results in a degenerate parition, a trivial set with only one (or a few) item(s) and another large set with all the remaining items, eventually requiring a linear number of partitions $O(n)$ to sort the data. The median of three technique helps speed up Quicksort back to log linear time $O(n \log n)$ for these situations. It works by picking 3 random pivot values from the current partition range and then choosing the median of these 3 values as the actual pivot. This greatly increases the probability that each pivot will partition the data into two large sets, resulting in a logarithmic number of partitions.

Median Tree: A left-balanced *k*d-tree in which the the *n* points belonging to a parent node are split almost exactly in half between the two child nodes. Given a current splitting axis, the median point is found, whose associated *cutting plane* separates the left and right child nodes into having $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor = n - \lceil n/2 \rceil$ points respectively. Informally, at most there is a difference in size of one point between the left and right child nodes.

Member Function: A function associated with an object-oriented *class* that is used to access or manipulate the objects specific data.

Memory Alignment: See *alignment*.

Memory Constraints: Various hardware limits on sizes, access speeds, transfer rates, cache-line sizes, etc. for each specific type of memory in a specific memory architecture.

Memory Hierarchy: A memory hierarchy in computer storage is a multi-level storage scheme which distinguishes each level in the hierarchy by its access speed. Each type of memory has many properties and performance characteristics of which the three most important are speed, size and price. Modern memory hierarchies typically have four main levels which themselves are often sub-divided into smaller hierarchies. 1) Registers 2) On-Chip Cache 3) On-line Main Memory (RAM), 4) Off-line file-based storage (hard drives). The top most level is usually the fastest, smallest, and most expensive. Each subsequent level is usually slower, larger, and less expensive than the previous level. The overall goal is to provide a memory system with amortized memory access speeds almost as fast as the fastest level but with an amortized cost per byte almost as low as the cheapest level of the hierarchy.

Memory Sub-system: The portion of a computer system (or chip) responsible for moving instructions and data between memory and the computational processing cores. Modern memory sub-systems often include connections to I/O devices such as graphics cards, disk drives, and networks. Modern memory sub-systems are typically implemented as a *memory hierarchy* which includes support for *caching*.

Memory Wall: A limit on parallel scalability. Memory *bandwidth* (and more generally communication bandwidth) and *latency* is not scaling at the same rate as computational bandwidth. See also *power wall*.

Metaprogramming: The use of one program to analyze, generate, manipulate, and/or transform other programs. See also *generic programming* and *template metaprogramming*.

Message Passing: See *remote message passing*.

Message Passing Interface: The message passing interface (MPI) is a standardized and portable remote message-passing system designed to function on a wide variety of parallel computers. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing parallel programs based on *remote message-passing* in different computer programming languages such as Fortran, C, C++, and Java. This standard has fostered the development of a parallel software industry, and encourage development of portable and salable large-scale parallel applications.

Method: See *member function*.

MIMD: A category in *Flynn's taxonomy* where processing occurs using multiple data streams and multiple instruction streams in parallel. This model supports task-level parallelism. MIMD is inherently more flexible than SIMD and thus more generally applicable to any algorithm. But MIMD is also inherently more expensive than SIMD as well. Beowulf clusters, Clouds, and Grids are all examples of MIMD computers that exploit request-level parallelism.

Minimal *k*d-tree: A minimal *k*d-tree uses exactly *n* points re-ordered as *k*d-tree search nodes to represent the *n* points contained in a search set *S*. Note: This doesn't use a linear *O*(*n*) amount of points but exactly *n* points.

MISD: A category in Flynn's taxonomy where a single data stream is executed in parallel on multiple instruction streams. This category is seldom used. No commercial architecture has been built using this model to date. The only partially related example I can think of is that the Space Shuttle executes three similar programs on three different computers which then vote (majority wins) in order to hopefully avoid a catastrophic software or hardware bug in any one system crashing the shuttle.

Monoid: An associative operator in some domain that has an *identity* element.

Moore's Law: Describes a long-term trend that the number of transistors that can be incorporated inexpensively on an integrated circuit chip appears to double approximately every 2 years. It is named for

Intel co-founder Gordon Moore who first described this trend in a 1965 paper. This law is often loosely (and incorrectly) used to describe any form of computational improvement that appears to grow exponentially over time.

Motif: Sometimes used as a synonym for *pattern*.

MPI: See *message passing interface*.

Multi-computer Machine Model: An abstract model of a parallel computer architecture for programmers. Under this model, a parallel computer is composed of several smaller computers. This model generally refers to an architecture in which each processor has its own stand-a-lone memory as compared to multiple processors with a shared memory. Contrast with the *von Neumann machine model*. See also *multi-core*, the *RAM machine model* and the *PRAM machine model*.

Multi-Core: A processor with multiple parallel cores. Each core supports execution of at least one hardware thread. Typically these multiple cores have a *shared memory*.

Multi-Core Era: Time period after which chip designs shifted away from frequency scaling and shifted into adding more parallel cores onto each chip. This era started in 2005 after chip architectures hit a *power wall* in 2004. See also *Frequency scaling era*.

Multi-Issue Execution: A type of ILP, also known as *super-scalar*, where each core can dispatch more than one instruction per clock cycle. Multiple independent instructions are dispatched and executed in parallel on multiple redundant functional processing units (ALU's & FPU's). This is similar in concept to having multiple assembly lines on one factory floor. The instruction scheduler must be able to detect dependencies between instructions and only schedule multiple independent instructions when safe to do so. See also *pipelining*, *out-of-order execution* and *simultaneous multi-threading*.

Multi-ported Memory: Multi-ported memory allows multiple reads/writes to occur in parallel when accessing the memory system for faster access and transfer speeds.

Multi-threading: With multi-threading, also known as *thread-level parallelism*, multiple active threads from a pool of thread are mapped onto and executed on multiple cores in parallel. Multi-threading can also be executed on a single-core system using time-slice multiplexing (multi-tasking) to switch between threads frequently. This results in logical concurrency not true parallelism but if the switching is fast enough it will appear to an end-user that multiple applications are running in parallel on the machine. Multi-threading with more threads than cores enables *latency hiding* for better system performance by avoiding idle waiting.

Mutex: Three similar definitions: 1) Short-hand for */mutual exclusion/*. 2) Synonym for a *lock* object. 3) A specific type of lock object.

Mutual Exclusion: With Mutual exclusion, Gate keeper objects *serialize* access to a section of code and/or data objects. Mutual exclusion includes objects such as Mutex's, semaphores, and atomic instructions. Protected code must be carefully written to ensure serialized access across parallel threads while avoiding concurrency problems like *deadlock*, *livelock*, and resource *starvation*.

Nearest Neighbor Search: With a nearest neighbor (NN) search, each query point is matched to the closest point (object) in a point cloud under some distance metric (such as Euclidean distance). This is an important problem for many different areas of computer science, including computer graphics, machine learning, pattern recognition, statics, and data mining. Specific types of NN searches include *Query Nearest Neighbor* (QNN), *k Nearest Neighbor* (kNN), *All Nearest Neighbor* (All-NN), *All k-nearest neighbors* (All-*k*NN), *Range Query Nearest Neighbor* (RNN), and *Approximate Nearest Neighbor* (ANN).

Nesting Pattern: A pattern that supports the ability to hierarchically compose other patterns.

Nested Reduce: A GPU implementation pattern that combines *Serial Reduce* & *Warp Reduce* building blocks in a nested manner to reduce a large run of *n* elements to a single total sum.

Nested Scan: A GPU implementation pattern that combines *Serial Scan*, *Warp Scan*, & *Run Update* building blocks in a nested manner to scan a large run of *n* elements.

NN: See *Nearest Neighbor Search*.

Node: 3 different definitions: 1) A node is a stand-a-lone "computer in a box". Each node usually consists of multiple processing cores, accelerators (GPUs), storage, and network support. Nodes are often networked together to comprise a cluster, cloud, grid, or super computer. 2) A node is a data structure containing a data element plus one or more links (pointers or references). A node can be used to create lists, trees, and graphs. 3) An axis-aligned geometric cell used within a *spatial partitioning data structure*, such as a *kd-tree*.

Non-blocking Algorithm: A *non-blocking algorithm* ensures that threads competing for a shared resource do not have their execution indefinitely postponed by mutual exclusion. In modern usage, an algorithm is non-blocking if the suspension of one or more threads does not stop the progress of the remaining threads. These algorithms are often designed to avoid using *lock*'s and often use simple *atomic operations*, such as *compare-and-swap*, instead. See also *lock-free* and *wait-free* algorithms.

Non-deterministic: Exhibiting a lack of deterministic behavior or ordering. So output results can vary from run to run of a non-deterministic program using the same input each time. Contrast with *deterministic*.

Non-Uniform Memory Access (NUMA): A multi-level memory system where different levels of the memory sub-system have different access rates. This results in varying performance for a program depending on where the data is physically located. Most modern computers are NUMA.

NO-OP: See NOP.

NOP: A NOP, or NO-OP, is a special do-nothing instruction. The instruction proceeds through an instruction processing unit (*pipelined*, *out-of-order*, *multi-issue*, *multi-threaded*, …) without doing any useful work or causing any side-effects. This type of instruction is useful as it is often used by hardware (or compiler) schedulers to delay execution (by *stalling*) for correct execution behavior in the presence of various *hazards* caused by parallel instruction processing. However, each NOP issued wastes a machine cycle which results in core *under-utilization*.

***nWarps*:** The ‹*nWarps*› template parameter represents *thread-level parallelism* on warp-threaded hardware. This parameter allows experiments on TLP to find the best performance. The programmer must use generic programming techniques to support a differing number of thread warps in a small range, typically ‹*nWarps*=[1-8]›, See also *nWork*, *WarpSize*, and *data block*.

***nWork*:** The ‹*nWork*› template parameter represents *instruction-level parallelism* on warp-threaded hardware. This parameter allows experiments on ILP to find the best performance. The programmer must use generic programming techniques to support a differing number of work items per thread in a small range, typically ‹*nWork*=[1-8]›, See also *nWarps*, *WarpSize*, and *data block*.

Objects: Objects are a language construct that associate a group of data fields with the corresponding code (functions) intended to act on and manage that group of data. Multiple functions may be associated with an object and those functions are called the *methods* (or *member functions*) of that object. *Objects* are considered to be members of a class of objects. And classes in turn can be arranged in a hierarchy in which subclasses inherit and extend the features of superclasses. That state of an object may or may not be directly accessible; in many cases access to an objects data fields may only be permitted through its methods.

Object-Oriented Programming: Object-oriented programming (OOP) is a programming paradigm based on *objects*, which contain both state (data described by data structures) and behavior (methods implemented as code). Each object is a specific instance of a *class* which describes both the data structure and methods. A class is an *abstract data type* with support for *polymorphism*, *inheritance*, *encapsulation*, and *information hiding*.

Observed Speedup: The speedup of parallel code over the equivalent serial code for solving the same problem, which is defined as

$$Observed\ Speedup = \frac{Time_{serial\ execution}}{Time_{parallel\ execution}}$$

Occupancy: The ratio of actual thread warps (known as active warps) that are tracked and scheduled on an *SM*(*X*) core to the theoretical maximum number of thread warps that could be tracked and scheduled on an *SM*(*X*) core for a given GPU architecture and GPU Kernel. *Occupancy* = (*#ActiveWarps* / *#MaxWarps*).

Occupancy Constraints: Various hardware constraints (number of registers, shared memory allocations, etc.) that limit the number of active warps on an SM(X) core, thus limiting occupancy.

Octree: An octree is a hierarchal *spatial partitioning data structure* used to speed up geometric searches. Each tree node has exactly eight children. Octree typically are used to partition 3D space by recursively dividing the starting bounding cube into 8 octant cubes. See also *BSP tree*, *quad-tree*, and *kd-tree*.

Offload: Placing part of a programs computation on an attached device such as a GPU or co-processor.

Offline: An algorithm which runs independently of the main CPU. For instance on an accelerator such as a GPU or co-processor.

Offside Node: In a *kd-tree nearest neighbor search*, the offside node is the left or right child node that does not contain the current query point of interest. Contrast with *onside node*.

One-to-One: A mapping relationship where a source object is related to only one destination object in a set, table, or array of parallel objects. See also *cardinality*, *one-to-many*, *many-to-one*, and *many-to-many*.

One-to-Many: A mapping relationship where a source object is related to many destination objects in a set, table, or array of parallel objects. See also *cardinality*, *one-to-one*, *many-to-one*, and *many-to-many*.

One-to-Several: A special case of a mapping relationship where a source object is related to a small number of destination objects in a set, table, or array of parallel objects. See also *cardinality*.

Online: An algorithm which can begin executing before all of its input data has been read.

Onside Node: In a *kd-tree nearest neighbor search*, the onside node is the left or right child node that contains the current query point of interest. Contrast with *offside node*.

OpenACC: Open Accelerators (OpenACC) is a programing standard for parallel computing developed by Cray, CAPS, Nvidia, and PGI. The standard is designed to simplify parallel programming of heterogeneous CPU/GPU systems. OpenACC is similar to OpenMP and will be merged into a future release of OpenMP.

OpenCL: Open Computing Language (OpenCL) was originally a standard defined by the Khronos group for supporting parallel and vectorized computations on graphics processors and attached co-processors. However, OpenCL has also been extended to specify parallel and vectorized computations on multicore host processors as well.

OpenMP: Open Multi-Processing (OpenMP) is an API that supports multi-platform shared memory multi-processing programming in the C, C++ and Fortran programming languages. It is available on most processor architectures and operating systems (Solaris, AIX, HP-UX, Linux, Max OS X, and Windows). It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. OpenMP uses a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the standard desktop computer to the supercomputer. OpenMP is run as a nonprofit technology consortium that includes many major computer hardware and software vendors (including AMD, IBM, Intel, Cray, HP, Fujitsu, NVIDIA, NEC, Red hat, TI, Oracle, and more…).

Optimization Techniques: A collection of programming (or compiler) techniques meant to improve the performance of a section of code, a function, a class, a module, or an entire program. See also *loop optimizations*, *data-flow optimizations*, and *inlining*.

Out-of-order Execution: With out-of-order execution, the hardware is allowed to re-organize the original program (instruction stream) order. This is allowed as long as re-arranging the order of instructions will not change the correct behavior of the program as compared to an *in-order* execution. Advanced hardware ILP techniques such as *score-boarding* and *Tomasulo's method* implement out-of-order execution for more efficient utilization of the processing cores. Software compilers can build instruction dependency graphs as *DAG*s and then perform a *topological sort* to find valid instruction re-orderings for better ILP performance. Contrast with *in-order execution*. See also *pipelining*, *multi-issue execution*, and *simultaneous multi-threading*.

Output-Dependency: See *write after write (WAW)*.

Over-decomposition: A parallel programming style where many more tasks, work-items, or data runs are specified than there are worker threads to execute them. On the positive side, this can benefit *load balancing* and *latency hiding*. On the negative side, this can result in increased *parallel overhead*.

Over-subscription: A parallel programming style where many more threads (or warps) are scheduled to run than there than there are processing cores to support them. On the positive side, this can help with *latency hiding* as individual threads (warps) stall. On the negative side, this can result in increased *parallel overhead*.

Pack Pattern: A data management pattern where certain elements of an array are discarded and the remaining elements are placed in a contiguous sequence, maintaining the relative order of the original sequence. Compare with the *expand pattern*.

Pad and Rake: A technique for mitigating *bank conflicts*. *Warp Padding* adds an extra data element (column) before or after each data warp (32 elements). This works well for sequential data runs that are a power of two in length, *k*=[2,4,8,16,32]. *Run padding* adds an extra data element (column) before or after each individual per-thread run. This works well for sequential data runs that are even in length, *k*=[6,10,12,14,18,20,…,30]. *Raking* changes indices to skip over the extra pad columns during indexing operations. Note: Sequential runs which are odd in run length, *k*=[1,3,5,7,9,11,…,31] do not cause bank conflicts and thus do not require the Pad and Rake technique..

Padding: Extra unused pad columns are inserted into shared memory arrays to prevent *bank conflicts* when loading short sequential runs for each thread in a warp. This works quite well for runs that are powers of two [2,4,8,…,32]. Padding requires *raking* to skip over the extra pad columns during indexing operations. See *run padding* and *warp padding*.

Page: The granularity at which virtual to physical address mapping is done in virtual memory management. Within a page, the mapping of virtual to physical memory addresses is continuous.

Page Thrashing: Page thrashing, also known as TLB Thrashing, occurs when a program spends most of its time loading and reloading virtual pages into physical memory instead of getting useful work done. This is typically caused by a large program (larger than can fit into physical memory) with a frequently accessed working set of active pages which get mapped from their own unique virtual addresses onto the same physical addresses. The O.S. keeps ping-ponging between evicting and reloading a small set of 2 or more logical pages onto the same physical page when instructions (or data) in each individual page are referenced.

Parallel: Physically happening simultaneously. Two or more threads (or tasks) that are all actually doing work that overlaps at some point in time are considered to be operating in parallel. When a distinction is made between *concurrent* and *parallel*, the key is whether at any point in time the work was done simultaneously. Multi-tasking operating systems have supported concurrency for decades via multi-threading even when simultaneous execution was impossible because there was only one processing core.

Parallel Algorithm: With *Parallel Algorithms*, the *depth efficiency* is logarithmic, $D(n) \approx O(\log n)$. The programmer typically deals with a divide and conquer type problem where there is a need to coordinate work across parallel threads during the individual steps of the parallel algorithm.

Parallelism: With parallelism, a programmer or system uses multiple resources concurrently to solve problems to increase performance and/or throughput. There are many types of parallelism including *Instruction-*, *Thread-*, *Task-*, and *Data-Level parallelism*.

Parallelization: The act of transforming serial code into parallel code. The parallelization of a program allows at least parts of it to execute in parallel.

Parallel Overhead: The amount of time spent coordinating parallel threads (or tasks) as opposed to doing useful work solving the original problem. Parallel overhead can be caused by factors such as creating threads, data communications between threads, synchronizing execution between threads, and terminating threads. In addition, extra overhead can be imposed by parallel compilers, libraries tools, operating system, etc.

Parallel Data Management Patterns: Parallel programming patterns for managing data. Examples of parallel data management patterns include – *gather*, *scatter*, *pack*, *pipeline*, and *geometric decomposition* including (stencil, partition). See also *serial data management pattern*.

Parallel Pattern: Programming patterns arising specifically in support of parallel computations. Examples of parallel patterns include the *map pattern*, the *reduction pattern,* the *scan pattern*, the *fork-join pattern*, and the *partition pattern*.

Parallel Program: A parallel program consists of multiple work-items (*tasks* or *runs* of data) executing in parallel on multiple cores.

Parallel Slack: The amount of "extra" parallelism available above the minimum necessary to use all the available parallel computing resources. See also *over-decomposition* and *over-subscription*.

Parameter: A value, reference to a value, or a pointer to a value(s) that is passed into a function, procedure, subroutine, command or program that represent input element(s) that need to be processed;

represent control elements that signal how other parameter(s) should be processed; and/or represent output element(s) where processed results should be stored. See also *by value*, *by pointer*, and *by reference*.

Partition Method: A sub-routine method used by both *Quickmedian* and *Quicksort*. Given an input set with a sub range [a, b], and a *pivot value* (*pv*) randomly chosen from the range of input elements. The partition method distributes all the elements in the range [a, b] into three data sets {*Left*: all elements less than *pv*, *Middle*: all elements equal to *pv*, and *Right*: all elements greater than *pv* }. This method takes linear time $O(n)$.

Partition Pattern: A pattern that decomposes the computational data domain for an algorithm into a set of non-overlapping subdomains called runs, tiles, or blocks. *Runs* are typically 1D, *tiles* are typically 2D, and *blocks* are typically 3D. Input data and output results may use different partition patterns as appropriate. See also the *geometric decomposition pattern* which is similar but allows overlap between sub-domains.

Pass: For *Radix Sort*, given data elements represented by numeric keys from a range [0, m), with a fixed-size *radix* with *digits* in the range [0, d), then the full radix sort will take $k = \lceil \log_d m \rceil$ passes to fully sort, where each sorting pass (one pass per digit in the key) is done using a stable sort such as *Counting Sort*.

Pattern: A pattern is a recurring combination of data and task management, separate and distinct from any specific algorithm or data structure. Patterns are universal in they apply to and can be used in any programming system. Patterns are also known as dwarfs, motifs, and algorithmic skeletons. Patterns are not necessarily tied to any hardware architecture, programming language, or operating system. Though features of the architecture, language, or OS may make implementing patterns easier in that specific environment. See also *parallel pattern*.

PCIe bus: A peripheral bus supporting relatively high bandwidth and DMA, often used for attaching specialized co-processors such as GPUs and NICs to the motherboard.

Peak Performance: The performance level (throughput, bandwidth) that a computer is guaranteed never to exceed because of hardware limitations.

Permutation Scatter Pattern: A form of the *scatter pattern* in which multiple parallel writes to a single storage location are disallowed. This form of scatter is deterministic but can only be considered safe if collision are checked for and prevented.

Pipeline Delays: Short stalls in instruction processing caused by various pipeline *hazards*. The most common delay is caused by instruction dependencies (*RAW*).

Pipeline Parallel Pattern: A set of data processing stages connected in series, generally the output of one stage becomes the input of the next stage in the sequence. Each stage of the pipeline can be scheduled onto its own thread allowing the data elements in different stages of the pipeline to be processed concurrently. FIFO queues can be used to help smooth out performance differences between different stages of the pipeline. Many algorithms can be quite naturally described as simple pipelines. A pipeline with only *k* stages only *scales* naturally up to *k* threads. To scale beyond the number of pipeline stages, it is necessary to exploit other forms of parallelism within each pipeline stage.

Pipelining: Two definitions: 1) Pipelining divides tasks (or instructions) into multiple sequential stages. Work items are processing sequentially with the output from one stage becoming the input into the next stage similar to an assembly line. 2) Pipelining is a form of instruction-level parallelism. With pipelining, the chip architects breaks each computational instruction into multiple processing stages. Each heterogeneous stage is performed by different processing units. The data operands stream through the hardware pipeline much like an assembly line.

Pipelined Processor: A pipelined processor can execute up to *k* instructions in parallel using *k* instruction processing stages laid out in a sequential pipeline where the output of each stage becomes the input to the next stage. This is similar in concept to a factory assembly line. The MIPS architecture is a well-known example architecture with 5 heterogeneous instruction processing stages: Instruction Fetch, Instruction Decode, Execute, Memory, and Write-Back often abbreviated as IF, ID, EX, MEM, WB.

Pivot Value: Part of the *partition method* (used in *quickmedian* and *quicksort*), where an arbitrary value from the current partition range [*min, max*] is chosen as the pivot. The rest of the elements in the current partition range are then distributed into three data sets those less-than the pivot value, those equal to the pivot value, and those greater than the pivot value.

Point to Point Coordination: Involves two threads (or tasks) communicating with each other with one thread acting as the producer of data and the other thread acting as the consumer of data.

Point Location Problem: Given a *spatial partitioning data structure*, and a *query point* (*q*) to search for, the point location problem wants to find the geometric region (cell or node) within the data structure which actually contains *q*. This is a simpler search problem then *nearest neighbor search*.

Polymorphism: A polymorphic data type or object-oriented class is one whose operations (or methods) can also be applied to the data values of other different but semantically compatible data types or classes. There are many different ways of achieving polymorphism including *function overloading*, *generic programming*, and *inheritance* between parent and child classes. See also *object-oriented programming*.

POSIX Threads: POSIX threads, often referred to as Pthreads, is a POSIX standard API for creating and managing threads as well as creating and managing locks to manage concurrency issues. Implementations are available on many Unix-compatible operating systems such as FreeBSD, NetBSD, OpenBSD, Linux, Max OS X, Solaris. DOS and Microsoft Windows implementations also exist.

Power Wall: A limit to the practical clock rate (frequency) of serial processors caused by the non-linear relationship between power and switching speed which limits parallel scalability. Computational performance (throughput) doubled approximately every 2 years from frequency scaling alone until chip architectures hit a power wall around 2004. This power wall means that running transistors faster than ~4 GHz generates more heat than simple air flow cooling (IE fans) can safely dissipate. Since 2004, most processors run between [1..4] GHz as a maximum upper bound on frequency to avoid overheating. See also *memory wall*, *frequency scaling era*, and *multi-core era*.

PRAM Machine Model: An abstract extension of the serial *RAM machine model* to support parallel programming. Under this model, one assumes that parallel random access to any data element in memory (storage) can be done in a constant amount of time. Four variations model whether parallel reads and writes from/into memory are exclusive or concurrent, only 3 variations are actually used in practice – 1) Exclusive Read Exclusive Write (EREW), 2) Concurrent Read Exclusive Write (CREW), 3) Concurrent Read Concurrent Write (CRCW).

Pragma: A form of program markup used to give hints or directions to a compiler but not change the semantics of a program written in a particular programming language. Also known as a compiler directive.

Precise Exception: In a pipelined (or parallel) instruction processing architecture, A precise exception means that a hardware exception has just occurred and all instructions in the stream before the exception have been successfully executed and all instructions after the instruction that caused the exception have not yet been executed. Modern processing cores require support for precise exceptions to support virtual memory. To handle precise exceptions, the processing core must commit changes in the original sequential program order. Many pipelined architectures make exceptions precise by *cancellation*.

Precision: The detail to which a number (typically floating point) is expressed. Lack of precision is the source of rounding errors in computations. The fixed and finite number of bits used to store a number requires some approximation of the true underlying value. These errors accumulate as multiple computations are made to the data in operations such as reductions and scans. Precision is measured in term of the number of digits that contain meaningful data, known as significant digits. Significant digits in computer science are often measured in bits because most floating-point arithmetic is done in radix-2. Radix-10 arithmetic should be expressed in terms of decimal digits when used.

Pre-computation: Precomputation is an optimization pattern where the programmer writes code that precomputes a value (or range of values) ahead of time which can then be stored in a look-up table and then reused at runtime for faster performance. For example, one might precompute the cosine of an angle at 100 separate angles between $[0, \pi/2)$ and then linearly interpolate between two bounding angles around the requested angle to get a close approximation to the actual results. This is a classic time-space trade-off and also a performance-accuracy tradeoff.

Predication: GPUs support conditional branching across thread warps using *serialization* via predication. *Predication* supports a special active/inactive flag across all 32 threads in a warp for all ISA instructions. Only active threads will actually execute the predicated instruction. Inactive threads will suppress any hardware results, effectively treating the instruction as a NOP instead. Branch outcomes {*true*, *false*} are used to set the predicate warp flags on a per-thread basis when executing the {*true*} and {*false*} sections of code associated with a branch condition.

Pre-emptive Multi-tasking: With pre-emptive multi-tasking, The scheduler (hardware or software) tries to guarantee each thread a regular "slice" of processing time in order for each thread to make forward progress on its own execution. The system scheduler may pre-empt any thread at any time by pausing it, and context switching to another active thread of execution. Pre-emption is usually done using an interrupt mechanism supported by the hardware (and/or operating system). Most modern operating systems support pre-emptive multi-tasking for multi-threading. See also *cooperative multi-tasking* and *time-slice*.

Pre-fetching: A programming optimization technique for increasing memory or I/O performance by fetching data shortly before the code actually needs it so that it will already be in faster cache memory when we actually do need it. This is a form of *latency hiding*. Pre-fetching requires a predictable access pattern for programmers to take advantage of. Care must be taken not to pre-fetch too much data otherwise it may be evicted by the time we actually try to use it. Too much pre-fetching could even lead to decreased performance due to *cache thrashing*.

Principle of Locality: See *locality*.

Principle of Mathematical Induction: Proving a predicate true for all cases can be done using the principle of mathematical induction as follows: Let $P(n)$ be a predicate, where n is an arbitrary positive integer. Suppose we can prove the following two statements. 1) *Base Case:* $P(n)$ is *true* for all $n \leq n_0$, where n_0 is a small integer (typically one). *2) Inductive Step:* Whenever $P(k)$ is *true*, it follows that $P(k+1)$ is also *true.* Then it follows that $P(n)$ is *true* for all positive integers n. See also *induction*.

Priority Scatter Pattern: A deterministic form of the scatter pattern in which an attempt to write multiple values in parallel to a single storage location results in one value (and only one value) being stored in the location. The value picked is based on some priority function, with all other values being discarded. The unique priority given to each parallel write in a priority scatter can be assigned in such a way that the result is deterministic and equivalent to a serial implementation.

Problem: A question, statement, or matter which seeks some *solution*. The problem is often uncertain or difficult. For mathematical problems, the solution is provided using a step by step application of mathematical operations resulting in a set of concrete numbers or an abstract formula. For software problems, the problem is often described in terms of transforming some set of input data into another set of output results. The software solution is provided using step by step computations encapsulated as a *program*. See also *algorithms*, *design patterns*, and *implementation*.

Process: An application-level unit of parallel work. A process has its own thread of control is typically managed by the operating system. Usually, unless special coordination logic is written into each program, a process cannot access the memory of other processes.

Producer-Consumer: A relationship where the producer creates data to pass to the consumer which consumes it in order to do further processing. If data is not consumed immediately on production, it must be *buffered*. See also *point to point coordination*.

Program: A program, also called an executable or binary, is a sequence of machine instructions, written to perform some specified task to solve a *problem* using a computer. The program is usually written by programmers as human readable source code in some higher level language and then compiled and linked into the actual machine language program targeted at a specific computer architecture. The program can then be run on a computer to solve the original problem. See also *ISA*, *solution*, *algorithm*, *design pattern*.

Programming: Computer programming -- also known as coding, software development, or software engineering -- is the iterative process of writing and editing source code to solve some set of tasks or problems. The process involves designing, writing, testing, debugging, profiling, and maintaining the source code for computer programs. Designing and writing source code often requires expertise in programming languages, formal logic, algorithms, data structures, and specialized knowledge of the application domain. Good programmers follow fundamental design properties when developing their code. These properties include – correctness, robustness, usability, readability, portability, maintainability, and efficiency.

Pseudo-code: Pseudo-code is an informal language-agnostic high-level description in English of an *algorithm*, this description can then be used to implement the algorithm in any programming language. It uses the structural conventions of a programming language (selection, iteration, sequences, etc.) but is intended for human reading and understanding. Pseudo-code often omits minor details such as variable declarations, parameter lists, sub-routine listings, etc. Pseudo-code often substitutes brief one line descriptions of what the code should do instead of the actual code. For example: "Set array to zero".

Pthreads: see *POSIX threads*.

Pure Function: A function whose output depends only on its input and that does not modify any other system state (I.E. the function doesn't cause any side effects).

QNN: See *Query Nearest Neighbor*.

Quadtree: A quadtree is a hierarchal *spatial partitioning data structure* used to speed up geometric searches. Each tree node has exactly four children. Quadtrees typically are used to partition 2D space by recursively dividing the starting bounding rectangle into 4 quadrant child rectangles. See also *spatial partitioning data structure*, *octree*, and *kd-tree*.

Query Nearest Neighbor: A type of *Nearest Neighbor Search* (NN) where each query point in a query set Q is matched to the closest point (k=1) in a search set S under some distance metric. See also *k*d-tree.

Query Point: A specific d-dimensional point for which to find the closest k neighboring points from some search set S containing n points. A spatial portioning data structure is often used to speed up the geometric search. See also *Nearest Neighbor Search*.

Query Set: A set Q containing *m query points* for which to find the closest k neighboring points from some search set S containing n points. A *spatial partitioning data structure* is often used to speed up the geometric search. See also *Nearest Neighbor Search*.

Queue: A basic data structure that stores data elements (commands, objects) as nodes in a list. Elements (nodes) are typically inserted at the back of the list and removed from the front of the list resulting in an FIFO processing order for data elements in the queue. See also *Deque* and *Stack*.

Quickmedian: A selection algorithm used to choose the k^{th} ordered point from a set of n unordered points. This is implemented using the partition sub-routine from quicksort. The entire algorithm takes linear time $O(n)$ on average and quadratic time $\mathrm{O}(n^2)$ in the worst-case.

Quicksort: An comparison-based sorting algorithm to efficiently sort unordered data. This is implemented using a partition sub-routine, the entire algorithm takes log-linear time $O(n \log n)$ on average and quadratic time $O(n^2)$ in the worst case.

Race Condition: *Non-deterministic* behavior in a parallel program that is generally considered a programming error. A race condition occurs when parallel tasks perform operations at the same time on the same memory location and at least one of the operations is a write. Code with a race condition may often operate correctly then mysteriously fail in unpredictable ways. Programmers can prevent race conditions by using *locks* to serialize access to the memory locations in contention. See also *concurrency issues*.

Radix: The base of a system of numeration. For example: a base 10 radix allows *digits* in the range [0,9) whereas a base 16 radix allows digits in the range [0,15). See also *Radix Sort*.

Radix Sort: With radix sort, an unordered integer sequence A is sorted into an ordered integer sequence S in k passes (with one pass per digit in the maximum key). There are two types of Radix Sort, least-significant-digit (LSD) and most-significant-digit (MSD). With LSD radix sort, on each pass, the n keys are counted and distributed into d sorted runs based on the current digit from each key. The d runs are then collated in monotonically increasing digit order, each pass is usually implemented using *Counting Sort*. Given m is the maximum key size, and d is the maximum digit value of the radix (typically d is much smaller than m, $d \leq m$) then the number of passes can be computed as $k = \lceil \log_d m \rceil$ or $k = \left\lceil \frac{B}{b} \right\rceil$, with B = the maximum bits per integer in a fixed-size range $[0, m)$, and b = the maximum bits per radix in a range $[0, d)$.

Raking: Raking is an indexing technique to skip over unused pad columns (see *padding*) inserted to prevent *bank conflicts* when working with short sequential per-thread data runs. The basic idea is to convert from a 1D index without padding (*stride* = *WarpSize* [32]) to a 2D index (warps & rows) and then convert back to a 1D index with padding (*stride* = (*WarpSize*+PAD) [33]).

Range Query Nearest Neighbor: A type of *Nearest Neighbor Search* (RNN) which returns all points from a search set S which are covered by each individual query region QR_i belonging to a query set QR (for Query Region). This causes a varying number of search results for each query region. Each individual query region is typically defined as a d-dimensional hyperbox or hyperball (of radius r_i).

RAM: Random-access memory is a form of computer data storage. A random-access device allows stored data to be accessed directly in any random order. Modern *DRAM* actually supports reading not as random machine words but in small parallel chunks called bursts.

RAM Machine Model: An abstract extension of the *von Neumann machine model* for programmers. Under this model, one assumes that random access to any data element in memory (which contains M elements) can be done in a constant amount of time $\Theta(1)$. Actual architectures take $O(k + \log M)$ time to access a sequential run of k elements in memory and $O(\log \log M)$ time to execute individual instructions in registers. To complicate matters further, modern computers actually use a multi-level *memory hierarchy* with varying access costs at each level and with support for *caching* at higher levels. Nevertheless, the constant time assumptions on memory access and instruction execution are standard and useful fictions for reasoning about algorithms on von Neumann machines. See also the *Multi-computer machine model* and the *PRAM machine model*.

Ramp-Down Time: Idling which happens when an instruction pipeline finishes up by draining out the last few instructions in the pipeline. See also *ramp-up time* and *idle cycles*.

Ramp-Up Time: Idling which happens when an instruction pipeline first starts filling up. An k stage pipeline takes at least k cycles to fill up fully. See also *ramp-down time* and *idle cycles*.

Reach-Back: An *inclusive scan* can be converted into an *exclusive scan* by *reaching back* one column in the scan results. For the first element in the inclusive scan (which has no previous column), the identity element ($\mathbb{I}$) should be substituted instead. For example, the inclusive scan under simple addition with identity $\{\mathbb{I} = 0\}$ resulting in $\{1,3,6,10\}$ can be converted into the corresponding exclusive scan as $\{\mathbb{I} = 0\}+\{1,3,6\} = \{0,1,3,6\}$

Recurrence Pattern: A sequence defined by a recursive equation. In a recursive equation, one or more initial terms are given and each further term of the sequence is defined as a function of the preceding terms. Implementing recurrences with recursion is often inefficient since the recursion tends to re-compute intermediate solutions that have already been computed on other recursion chains. A *loop* with dependencies between iterations can also be described as a recurrence. In the case of a single loop, if the dependence is *associative*, it can be parallelized using the *scan pattern*. If the dependence is inside a multidimensional set of nested loops with dimension *n*, the entire nested loop can always be parallelized over *n*-1 dimensions using a hyperplane sweep. In addition nested loops can often be parallelized using the *fork-join pattern*.

Recursion: The act of a function being re-entered while an instance of the function is still active in the same thread of execution. In the most common case, a function directly calls itself. Recursion is often supported by storing function state for each function instance on a stack. Bounding the depth of the recursion is important to avoid infinite recursion and unbounded stack growth. See *Recursion*.

Reduce: An operation applied to a collection of values to merge all values down to a single value. A simple example is computing a total sum on *n* numbers. This can be done in serial or parallel.

Reduce-than-Scan: A 3-level GPU implementation pattern on *n* elements where 1) Do a *serial reduce* on short runs of [*n*/*p*] elements per thread in parallel to generate run sums 2) Do an *inclusive scan* on the run sums and then *reach back* one column to get inclusive prefix sums for each run. 3) Do a scan on each run from step 1 and also do a *run update* by adding in the missing run prefixes from step 2 to each element in the run. This pattern requires 3 I/Os per data element on average to fully scan the data (1 read for the reduce (step 1), 2 (1 R + 1 W) for the scan (step 3). This parallel scan pattern is typically implemented using a hierarchical or *nested scan*.

Reduction Pattern: A *collective coordination* model, where all threads (or tasks) cooperate to merge *n* results down to a single result. The merging is typically done using binary operations to combine values in pairs. The binary merging operation should be *associative* in order to allow parallelization of the process. It is often also useful if the binary operation is commutative, which allows data re-ordering. Integer addition and multiplication is both associative and commutative. Floating-point addition and multiplication, however is not due to truncation and round-off errors. Using the reduction pattern on floating-point data may lead to non-deterministic results for parallel algorithms and different results between the serial and parallel implementations of the same algorithm. Though still useful, the user must be aware that the floating-point reductions are approximate and not exact.

Reduction Variable: A variable that appears in a loop for combining the results of many different loop iterations. For example, a variable that represents the total sum for a loop that computes the total sum of n values.

Re-entrancy: A re-entrant function is a function f that can be paused at any point in its execution, switch to another thread (which may or may not call the same function), and eventually resume execution and complete correctly. This means all function state (function parameters, stack variables, etc.) are *thread-local* and can be correctly stored/loaded through *context switching*. A re-entrant function should avoid static variables, global variables, or any other type of non-local state. Alternately, non-local data can be made re-entrant if all access to this data is protected using *mutual exclusion*.

Refactoring: Reorganizing code to clean it up, generalize it, or make it better suited for some new design purpose, such as enabling cross-platform compatibility or *parallelization*.

Registers: A very fast but usually very-small on-core set of memory locations meant to be used directly by each processing core for maximum speed. Registers can typically be read and written in a single machine cycle. Input operands and output results are temporarily stored in registers while being processed by machine instructions. Data is transferred between the core registers and memory storage using load/store instructions.

Register Pressure: The programmer has the illusion of allocating as many variables as they want as they write code. However during compilation, the compiler must decide how to allocate these variables onto a small finite set of registers. Register pressure is the term used when there are much fewer hardware registers available than would have been optimal to schedule all variables without any conflicts. Register pressure means that register reloads and register spills must be used to deal with the scheduling conflicts between conflicting variables resulting in lower performance than optimal.

Register Renaming: A technique for handling WAR and WAW data hazards caused by *aliasing*, which is the unnecessary sharing of a destination (register, memory) across multiple instructions in a pipelined architecture. This solution uses different destinations (registers) for the instructions in competition. *Tomasulo's method* eliminates both WAR & WAR hazards in hardware using dynamic *register renaming*.

Register Spills: Programs with too many variables may exceed the maximum number of registers allocated per-thread in concurrent thread blocks on each SM for a specific kernel on a GPU device. CUDA works around having register spills by using *Local memory*. Local memory is a technique where the extra variables are put into global memory and accessed at global memory speeds. This enables kernel compilation at the cost of relatively slow I/O for the affected variables.

Relative Speedup: Speedup in which a parallel solution to a problem is compared to a serialization of the same parallel solution. For instance, the parallel algorithm might be restricted to only using a single thread to serialize performance.

Remapping Array: Given an input array *S* with *n* points that has been re-ordered as *S*' (by sorting for instance). Create a secondary array *I* as the original indices [0, 1, 2, …, *n*-1] of *S*. Create a remapping array *I'* by re-ordering *I* in the exact same manner as *S'*. This allows original elements in *S* to be looked up from search results found in *S*' via the remapping array *I*'.

Rematerialization: Rematerialization is an implementation optimization, also known as re-calculating, where the programmer recomputes intermediate results from original inputs as often as needed. This optimization pattern assumes that recomputing the intermediate results from the original input is faster than computing it once, storing the result, and reloading it as needed. Contrast with *Precomputation* and *reusing*.

Remote Message Passing: A communication model where *distributed memory* is partitioned across different cores (or computer nodes) and the best way to access data on other cores (or nodes) is remotely using messages sent across a network. The message transfer speed is dependent on the distance between nodes, network topology, and the size of the message. Large messages are often broken into a set of smaller fixed size messages.

Replicate: Two definitions 1) To solve structural hazards in pipelined micro-architectures, architects often replicate functional units on a chip, IE make extra copies of functional units to eliminate competition between stages. 2) To increase parallelism, architects make many copies of simple chips distributed across a larger chip.

Request-Level Parallelism: A type of parallelism which divides mostly independent read-only requests across multiple computing nodes. Each request (task) is routed onto a node (server) which fulfills the request. Request-Level parallelism exploits parallelism among largely decoupled tasks specified by the programmer or the operating system. This type of parallelism is typical of webservers rendering webpages for customers surfing a company's web-site.

Resource Contention: Resource contention occurs when multiple parallel threads (or tasks) all attempt to access the same data element, memory address, or system object at the exact same time. Resource contention can result in race conditions. Resource contention can be solved by *mutual exclusion* or by careful *partitioning*.

Response Time: The time between when a request is made and when the response is received.

Re-using: Re-using is an implementation optimization technique where the programmer computes an intermediate result once, stores it in registers (or memory) and reloads it as needed for future re-use. This optimization pattern assumes that the stored value will be frequently reused (to take advantage of temporal locality) and that reloading the value is faster than recomputing it. Contrast with *rematerialization*.

RNN: see *Range Query Nearest Neighbor*.

Rotate Pattern: A special case of the *shift pattern* that handles boundary conditions by moving data that moves out of range on one side back around to the other side.

Run: Two different definitions: 1) A single execution of a program on a computer is often called a run. 2) A 1D sub-range of data from a larger data array. Each 1D run is typically contiguous and sequential. All runs typically partition the original array using the *partition pattern*. The set of runs are mapped onto an executed in parallel on multiple processing cores. The processing cores typically execute the same program when processing the data runs. See also *block, data-level parallelism, parallel program*, and *tile*.

Run Padding: With *run padding*, a single unused pad column is added after (before) each short data run to mitigate bank conflicts in shared memory. It works on runs with even run lengths {6, 10, 12, 14, 18, 20, 22, 24, 26, 28, 30} by effectively treating them as runs with odd run lengths instead. Runs with even lengths that happen to also be a power of two {2, 4, 8, 16, 32} should use *warp padding* instead as it requires less total padding (one pad column per data warp). Run indices must be raked (see *Pad and Rake*) to skip over unused pad columns.

Run Reduce: Run Reduce, also known as *reduce-reduce*, is a parallel pattern for reducing a long run of *n* elements to a single sum in two parallel stages. The reduction is done by combining pairs of elements using a binary sum operator. In the first stage, *n* elements are partitioned across *p* threads into runs of $\left\lceil \frac{n}{p} \right\rceil$ elements each. Each of *p* threads then serially reduces its assigned run to a single run-sum. In the second stage, a single thread serially reduces *p* run-sums to a single final sum. Compare with *Tree-reduce*.

Run Update: A building block for GPU Scan used to update a short run of *n* =[2-32] elements with a common prefix value under some associative binary summation operator. See also *Serial Reduce*, *Serial Scan*, *Warp Reduce*, and *Warp Scan*.

Safety: A system property that automatically guards against certain classes of programmer errors, such as race conditions.

Saturation: Saturation arithmetic has maximum and minimum values that utilized instead of the logical results when the logical results would be higher or lower respectively than the max/min values. Numerical representations on computers always are limited in precision and range. Saturation arithmetic is one way to ensure computations do not exceed a computers precision and range. Saturation arithmetic for signed values is not associative. Contrast with *wrap-around arithmetic*.

Scalability: A measure of the performance increase in a system as more and more parallel resources are added into the system.

Scalable: A program (or system) is scalable if its performance increases when additional parallel computing resources are added into the system. In the ideal case, a parallel program will demonstrate a *linear speedup* as more parallel computing resources are added into the system. Programs (and systems) often fail to achieve linear speedup due to *parallel overhead* or *under-utilization* of parallel resources.

Scalar Processor: Each instruction is executed one at a time on a single processing core. This is the essence of the SISD processing model.

Scalar Promotion: When a scalar and a vector of size *n* are combined under some operation, the scalar is automatically treated as a vector with *n* elements, with all elements set to the original scalar value. This approach allows the original operation to proceed as intended.

Scatter Model: A *collective coordination* model, where one thread (or task) sends different messages to all other threads (or tasks). The messages are typically data decomposed.

Scan: With scan, also known as *prefix sum*, an output sequence $S = [s_1, s_2, \cdots, s_n]$ is generated from the input sequence $A = [a_1, a_2, \cdots, a_n]$, where $s_i = \mathbb{I} \oplus a_1 \oplus a_2 \oplus \cdots \oplus a_{i-1}$ for *exclusive scan*, or $s_i = a_1 \oplus a_2 \oplus \cdots \oplus a_i$ for *inclusive scan*. In other words, the i^{th} output element is the total sum of the first *i* (or *i*-1) elements from *A*. The scan is done using a binary *associative* operator $\oplus$ with identity $\mathbb{I}$. A simple example is computing a prefix sum on n numbers under addition with zero as the identity. The scan can be done in serial or parallel. See also *Reduce*.

Scan Pattern: Pattern arising from a 1D recurrence relationship on some computation. This often arises as a loop-carried dependency where the computation of one iteration depends on the results from the previous iteration. Such loops can still be parallelized if the underlying dependency can be expressed using an associative operation.

Scan-than-Fan: A 3-level GPU implementation pattern on *n* elements where 1) Do a *serial scan* on short runs of [*n*/*p*] elements per thread in parallel to generate prefix sums and a run sum per run. 2) Do an *inclusive scan* on the run sums and then *reach back* one column to get inclusive prefix sums for each run. 3) Do a *run update* on each run from step 1 by adding in the run prefixes from step 2 to each element in the run. This pattern requires 4 I/Os per data element on average to fully scan the data 2 (1 R + 1 W) for the scan (step 1) and 2 (1 R + 1 W) for the run update (step 3). This parallel pattern is typically implemented using a hierarchical or *nested scan*.

Scan Update: A building block for GPU Scan used to scan a short run of *n* =[2-32] elements into a prefix sum under some associative binary summation operator and the update the entire run with a common prefix. Basically a *Serial Scan* is combined with a *Run Update*.

Scatter: A type of *map* primitive, which scatters elements from a small source array *S* into a large destination array *D* using an index map. Also see *Copy*, *Fill*, and *Gather*.

Scatter Pattern: A set of parallel random writes into memory. A scatter takes a collection of memory addresses and overwrites data at those memory addresses. Scatters are equivalent to random writes inside a *map pattern*. Given an input array of size m, an index array of size m, and an output array of size *n*, typically with $n \geq m$, each element of the input array overwrites some element in the output array at the given index. *Output*[*Index*[*i*]] = *Input*[*i*] for all $i \in [0, n)$. The scatter pattern is the inverse of the *gather pattern*. Unlike gathers, scatters must deal with collisions which occur when multiple input elements being processed in parallel all map onto the same output index. There are 4 common ways to resolve collisions -- *permutation scatter*, *atomic scatter*, *priority scatter*, and *merge scatter*.

Scoreboarding: A type of hardware *ILP* architecture that supports parallel *out-of-order execution* of instructions in an instruction stream. A scoreboard is a table maintained in the hardware that tracks all active instructions (status, assigned resources, registers, etc). The scoreboard is used to dynamically and safely schedule each instruction as soon as there are no conflicts and as soon as functional computing units (ALU's or FPU's) are available to take the instruction.

Search Pattern: A pattern than finds data from a larger collection that meets some criteria.

Search Set: A set *S* containing *n* search points that need to be searched for a closest points to a query set Q containing *m* query points. A *spatial partitioning data structure* is often used to speed up the geometric search. See also *Nearest Neighbor Search*.

Segmentation: A representation of a collection of data which is divided into non-overlapping subdomains. Typically, the various subdomains vary in size. *Reduction* and *scan patterns* can be generalized to operate over the segments of a collection independently while still being parallelized and load balanced across the original large domain.

Selection Pattern: A serial pattern in which one of two flows of execution are chosen based on a Boolean test predicate {*true*, *false*}. Contrast with *Speculative Execution*.

Separating Hyperplane: A *n*-1 dimensional plane that can be used to determine the sweep order for executing a *n*-dimensional recurrence in parallel.

Sequence Pattern: A serial pattern in which tasks are executed sequentially in-order one after the other. Each task completes before the next task starts. In some sense, this is the most fundamental programming pattern.

Sequential Consistency: Sequential consistency is a memory consistency model where every task in a concurrent system sees all memory writes happen in the exact same order, and an individual tasks own writes occur in the original order that the task specified.

Serial: A program, task, or system that is neither concurrent nor parallel. Typically there is a single processing core that executes a single program (instruction stream) sequentially in-order as it was original written.

Serial Bottleneck: A region of an otherwise parallel program that runs serially.

Serial Consistency: A parallel program that produces the same output results from the same input data as an equivalent serial program.

Serial Illusion: The apparent serial execution order of machine language instructions on a computer. In fact, modern processing cores support various hardware ILP techniques which run instructions in parallel and often re-order instructions for better performance. See *out-of-order*.

Serial Data Management Patterns: Serial programming patterns for managing data (allocation, sharing, reading, writing, copying, etc.). Examples of these serial data manage patterns include – *Random access*, *Stack Allocation*, *Heap allocation*, *Closures*, and *Objects*. See also *parallel data management patterns*.

Serial Patterns: Common programming patterns arising in support of serial computations. Examples of serial patterns include – *sequence*, *selection*, *iteration*, *recursion*, and *nesting*. See also *parallel patterns.*

Serial Reduce: A building block for GPU Reduce & GPU Scan used to reduce a short run of *n* =[2-32] elements down to a single total sum value under some associative binary summation operator. See also *Serial Scan*, *Warp Reduce*, and *Warp Scan*.

Serial Scan: A building block for GPU Scan used to scan a short run of *n* =[2-32] elements into a prefix sum under some associative binary summation operator.

Serial Update: Another name for *Run Update*.

Serial Trap: A serial trap is a programming construct that semantically requires serial execution for correct results. Even though a specific problem might be over-constrained with respect to concurrency by such semantics. The term "trap" acknowledges how such constructs can easily escape the attention of programmers as barriers to parallelism. In part, because these constructs are common and were not intentionally designed to prevent parallelism. For example, the **`for`** statement in the C language, has semantics that dictate the order of iterations by allowing each iteration to assume that all prior iterations have already been fully executed. These semantics make it harder for parallel compilers to extract natural parallelism from these **`for`** loops. Many loop constructs actually do not need to assume anything about the execution order of prior loop iterations for correct behavior. Unfortunately, most programmers use the **`for`** statement as a matter of course when writing loops in C or C++.

Serialization: Three definitions: 1) Refers to when the potentially parallel tasks in a parallel program are actually executed in a specific serial order. This is typically caused by resource constraints or locking mechanisms used in the program to prevent parallel *resource contention*. 2) A hardware scheduler may serialize access to a shared resource, such as memory, by parallel threads (cores) to ensure correct program behavior. 3) On GPUs, warp branching is supported by serialization, which means the program will execute both the {*true*} and {*false*} paths sequentially inactivating those threads for which the branch test doesn't apply. See also *branch divergence* and *predication*.

Set Associative Cache: A cache organization in which a particular location in main memory can be stored in a (small) number of different locations in the cache.

Several-to-one: A special case of a mapping relationship where a small number of source objects are related to a single destination object in a set, table, or array of parallel objects. See also *cardinality*.

Shared Address Space: Even if parallel processing cores do not share a common physical memory, they may agree on conventions that allow a single unified set of addresses to be used when addressing all memory. For example, one range of addresses could always refer to memory on the host CPU processor, while another range could refer to memory on a specific co-processor such as a GPU. The use of unified addresses simplifies memory management.

Shared Memory: Two different definitions: 1) A memory model where each parallel processing core has access to a shared bank of memory common to all cores. As a result of the all point-to-point connections required and the complex hardware required to maintain *cache coherence* across parallel cores this model typically is limited to 2-8 cores per shared memory bank. Programmers need to use some form of *synchronization* to prevent resource contention between parallel threads (or cores). 2) A specific form of GPU scratchpad memory local to each SM core used to store intermediate results for better algorithmic performance.

Shift Pattern: A special case of the *gather pattern*. The shift pattern translates (or offsets) the location of each element in the array by a fixed index offset. The basic shift pattern ignores (drops) values that are out of range. While the *rotate pattern* wraps out of range indices around to the other side of the array and stores them there.

SIMD: A category in *Flynn's taxonomy* where processing occurs using multiple data streams all sharing a single instruction stream in parallel. Intel's MMX instructions and GPUs are examples of SIMD computers. See also *MIMD*.

Simultaneous Multi-Threading (SMT): A form of ILP. SMT improves on multi-issue and out-of-order execution ILP techniques by adding direct support for multi-threading into the processing core scheduler. Instructions from more than one executing thread can be scheduled onto the multi-issue processing units on any given machine cycle. Since instructions from different threads are guaranteed to be independent this multi-threading scheduling helps each core stay busy doing useful work. See also *latency hiding*.

SISD: A category in *Flynn's taxonomy* where processing occurs using a single data stream and a single instruction stream. Most programmers think of this as the standard sequential computer as in the von Neumann model even though modern CPUs actually support both ILP and TLP.

SIMT: Single-Instruction Multiple-Threads. A variation on *SIMD* that also supports multi-threading in addition to vectorized instruction processing (SIMD).

Skeleton: Two Definitions: 1) A parallel implementation pattern. 2) A code framework, AKA as a stub, that compiles and runs but does no useful work. A programmer must fill in the missing details to solve a specific *problem*. The skeleton is meant as a good starting point for programmers. See *data access skeleton* (*DASK*) and *block access skeleton* (*BASK*).

SMT: See *simultaneous multi-threading*.

SoC: System on a chip (SoC). As CPU cores gain more transistors more and more functionality that used to be handled by specialized co-processors have been moved back onto the chip itself. This includes support for items such as Compression, Graphics, Audio, I/O protocols, Networks, etc.

Software Pipelining: Software pipelining is a loop optimization technique. Software pipelining is a type of out-of-order execution except the reordering is done by a compiler or a programmer instead of by ILP hardware. Here is a simple example:

```
for i=1 to n {
  A(i); B(i); C(i);
}
```

where C depends on B which depends on A. This loop can be rewritten as a software pipeline in batches of 3 as follows:

```
nBatches = n/3;  nLeftover = mod(n,3);
for (i=1; i<nBatches, i+=3) {
  A(i); A(i+1); A(i+2);  // Execute 3 A commands
  B(i); B(i+1); B(i+2);  // Execute 3 B commands
  C(i); C(i+1); C(i+2);  // Execute 3 C commands
}
// Handle [0-2] leftover data items here …
```

This technique reduces loop overhead and reduces data dependencies for better ILP. Special care must be taken to handle any partially left-over data items at the end of the loop. See also *loop unrolling* and *batching*.

Software Thread: A software thread is a virtual hardware thread – Multiple software threads are mapped onto a single hardware thread (or processing core) by the thread scheduler (hardware or software). Typically there are more software threads than there are processing cores. Fine-grained context switching is used to switch between software threads on each core to support *latency hiding*.

Solution: Four definitions: 1) The act of solving a *problem*, question, etc. 2) The answer to a problem, question, etc. 3) In mathematics, a step-by-step series of operations that solves a problem. 4) In computers, a section of code, function, module, or program that solves a problem.

Space Complexity: A complexity measure for the amount of memory (or storage) used by an algorithm as a function of problem size. See also *asymptotic analysis* or *big "O' notation*.

Span: See *Depth*.

Span Complexity: See *Depth Complexity*.

Spatial Locality: The frequent access of data elements which are relatively close to the original location of the first access into memory storage. See also *locality* and *temporal locality*.

Spatial Partitioning Data Structure: A data structure (typically hierarchical) used to organize points (or objects) in *d*-dimensional geometric space for faster search algorithms. See also *kd-tree* and *octree.*

Spawn: The creation of a new *thread* (or task) from within an existing *thread* (or task). See also *fork* and *join*.

Speculative Execution: A parallel form of the *selection pattern*. Where both paths in a conditional branch are executed in parallel using redundant processing units and once the branch condition outcome {*true*, *false*} is successfully known then the instructions from the invalid path are *cancelled*.

Speedup: Speedup is the ratio between the time it takes to solve a problem using a single processing core vs. the time for solving the same problem using *n* parallel processing cores.

Split Pattern: A generalized version of the pack pattern that takes an input collection and a set of Boolean labels {*true*, *false*} that go with every data element in that collection. It re-organizes the data so all elements marked with {*false*} are stored in one output sub-group (typically at the front) and all elements marked with {*true*} are stored in another sub-group (typically at the back). The deterministic version of this pattern is stable, so it preserves the same relative order of data elements form the original collection in each output sub-group.

SPMD: Single-Program, Multiple-Data, A program that runs a single function on multiple processing cores, but allows each function instance to follow different control flow paths during execution.

Stable Sort: A sort is *stable* if elements with equal *keys* maintain the same relative order. A more descriptive way of saying this is as follows: Given any two elements (a_i, b_j) with equal keys (a_i.*key* == *b*.*key*), such that *a* precedes *b* in the unsorted input ($i < j$) then after the sort is done, then *a* still precedes *b* in the sorted output ($i' < j'$). An *unstable* sort does not preserve the relative ordering of elements with equal keys.

Stack: A basic data structure where data elements are kept in a list (representing a pile of objects). Elements are both inserted and removed from the top of the list resulting in a *LIFO* processing order for elements kept on a stack. See also *Deque* and *Queue*.

Stall: Two related definitions: 1) A processing core which does nothing useful for many cycles is said to be stalled. 2) Each machine cycle where the instruction scheduler issues *NOP* “do nothing” command to prevent *hazards* from causing incorrect behavior is known as a stall.

Stalling: An ILP technique, also known as bubbling, for handling various *hazards* in a pipelined micro-architecture. With stalling, the scheduler (hardware or software) inserts *NOP*’s (do nothing instructions) into the pipeline until the hazard is resolved. This gives correct results at the cost of *under-utilizing* the processing core.

Standard Library: The C++ standard library is a collection of classes, algorithms, and functions. The C++ standard library provides generic containers; functions to use and manage those containers; function objects; generic strings; I/O streams; and many other useful algorithms and tasks. The C++ standard library currently follows conventions introduced by the original *Standard Template Library* (STL) and has been influenced by research in generic programming. The C+ standard library and STL overlap in many features and functionality but neither is a strict superset of each other.

Standard Template Library: The C++ Standard Template Library (STL) is a software library for the C++ programming language that provides four main components – algorithms, containers, functional, and iterators. The STL achieves all of its components through the use of C++ templates. This provides compile-time *polymorphism* that is much more efficient than traditional run-time polymorphism. The four main ideas behind the STL are – *generic programming*, *abstractions* without loss of efficiency, the *von Neumann computation model*, and *value semantics*.

Starvation: A concurrency issue where one or more threads fail to make forward progress towards completion because they are perpetually denied necessary system resources. Without those resources, the thread can never finish its assigned task. This is typically caused in a multi-threaded environment where the thread manager (or scheduler) implements priority pre-emption incorrectly. See also *deadlock* and *livelock*.

Static: A static algorithm (or data structure) does not allow any insertions, deletions, or any other modifications to the original input data set (or data structure once built) until the algorithm has completed. Contrast with *Dynamic*.

Static Scheduling: With static scheduling, the thread scheduler (typically software) pre-computes a thread schedule before executing a program and then sticks to the thread schedule during the entire execution of the program. This is typically implemented as part of a compiler. This approach can result in poor overall performance as the compiler must be strictly conservative in how it assumes pointers alias memory accesses. Often threads idle waiting for their turn, get woken up only to promptly go back to sleep while waiting on slow I/O resources to complete.

Stencil Pattern: A regular input data access pattern based on a set of fixed offsets relative to an output location. The stencil over the input is repeated for every output position in the output grid. This pattern can be seen as combine the map pattern with a local gather over a small neighborhood in a fixed pattern of offsets. Some care must be taken at the boundaries of the grid for how to deal with out of range offset indices. Stencil patterns are common in algorithms that deal with regular grids of data, as in image convolution.

Step: A single step in a chain of processing steps that make up the critical path through a task dependency graph for a parallel program. The *depth* is a measure of how many steps are in this critical path.

Step Complexity: See *depth complexity*.

Strangled Scaling: A programming error in a parallel program where performance is poor due to high resource contention between threads or high parallel overhead. The performance may be so bad that the parallel algorithm may underperform the serial version of the same algorithm.

Stream: A sequence of data elements made available over time. A stream can be thought of as a conveyor belt that allows data elements to be processed one a time sequentially instead of in larger batches.

Streaming Algorithms: Streaming algorithms are algorithms for processing data streams in which the input is presented as a sequence of data elements which can only be examined in a few passes (typically just one). Streaming algorithms are usually constrained by the access pattern, memory, and time. The access pattern is typically sequential, in other words, the data elements must be processed one at a time as they are received. These algorithms typically have limited memory available to use for processing data (typically much less than the actual input size) and also limited processing time per data item in the stream. These constraints mean that streaming algorithms are often used to create summaries or smaller datasets which approximate the full data stream. See also *BYTE stream*, *RAM model*, *map pattern*, *pipeline pattern*, *producer-consumer relationship*.

Stream Processing: A paradigm for computer programming for SIMD architectures. Given a set of data (the stream), a series of operations (*elemental functions*) is applied to each element in the stream. These elemental functions are often arranged in a pipelined manner. To reduce I/O operations these pipelined functions are often fused into one larger uniform function. Stream processing works best for programs that exhibit three characteristics – High *Compute intensity*, *Data Independence*, and *Data locality*. Compute Intensity means that the ratio of computations to communications is quite high. *Data Independence* in this context means that each data element can be processed independently from any other element. *Data locality* means that data is produced (typically only once), read once (or twice) during the processing, optionally output (again typically only once) and never accessed again. GPUs are examples of high throughput stream processors. See also *Map Pattern*.

Strength Reduction: Strength reduction is a *data-flow optimization* where expensive (or slow) operations are replaced with equivalent less expensive (or faster) operations. For instance computing an index inside a loop for array access using "strong" multiplication can be replaced by "weaker" addition instead. Another example, replacing division by a constant by multiplication using its reciprocal. See also *constant folding*, *code fusion*, and *dead code elimination*.

Strip-Mining: When implementing a stencil or map pattern, an optimization that groups instances in a way that avoids unnecessary and redundant memory accesses and aligns memory accesses with the vector lanes of the SIMD cores.

Strong Scalability: A form of scalability that measures how performance increases when adding in additional parallel processing resources but leaves the original problem size fixed.

Strongly Serial Algorithm: With *Strongly Serial* algorithms, the *depth efficiency* is linear, $D(n) \approx O(n)$. With these algorithms, there is no obvious way for a programmer to divide work across multiple parallel cores, the entire algorithm is almost entirely one long chain of dependent steps.

Structural Hazard: A structural hazard occurs when a specific part of the processor's hardware needs to be used by two or more stages of an instruction pipeline concurrently. For example, A pipelined architecture might only have a single memory unit that is used both in the fetch stage to retrieve instructions from memory and also in the memory stage where data needs to be transferred (read/write) between registers and memory. Architects often solve structural hazards by *replication* -- creating multiple redundant processing units (one per stage) to eliminate competition.

Structure-of-Arrays (SoA): A data layout for collections of data elements where each data element is a structure made up of heterogeneous fields. For data storage, all the data for the first field in each data element is stored in one contiguous array and all the data for the second field in each data element is then stored in another contiguous array. This contiguous storage layout is repeated for all fields in data structure. Compare with *array-of-structures (AoS)*.

Successor Function: In a *fold*, the function that computes a new state given the old state and a new input data item.

Supercomputing: See *high performance computing*.

Superlinear speedup: Speedup where performance grows at a rate greater than the new parallel processing resources are added into the system. Since linear speedup is optimal for parallel algorithms, the superlinear speedup is usually the result of some additional performance optimization coming into play as a result of the new parallel layout of the algorithm (better caching opportunities, reduced branching, etc.)

Super-Scalar: See *Multi-issue*.

Super-Scalar Processor: A super-scalar processor executes more than one instruction per clock cycle by simultaneously dispatching multiple independent instructions across multiple redundant functional units (ALU's, FPU's) on each processing core.

Superscalar dependency pattern: A sequence of tasks (instructions) ordered only by true data dependencies rather than a sequential ordering of the tasks. This allows the parallel execution (multi-issue & out-of-order) of tasks (instructions) that have no dependencies on each other.

Switch-on-event Multithreading: A technique that supports the execution of multiple threads on a single core by context switching to another thread on long-latency events such as cache misses when transferring data between main memory and registers.

Symmetric Multi-processor (SMP): A hardware architecture where multiple typically homogenous processing cores share a single address space and access to all system resources. See *shared memory computing*.

Synchronization: The coordination of parallel threads (or tasks) in real time to ensure the correct behavior of a parallel algorithm across all threads (or tasks). This is often implemented by establishing a synchronization point within the code where a thread (or task) must wait until all other threads (tasks) in the same group (warp or block) reach the same synchronization point. Synchronization usually involves waiting by one or more threads (or tasks) and can thus slow parallel performance.

Synchronous Coordination: Involves multiple threads (or tasks) trying to communicate data or coordinate work between themselves. Threads are now dependent on each other while doing coordination. All threads involved in transferring data must wait until the transfer is complete before proceeding on to other work. This results in gaps of idle work spent waiting instead of doing useful computations. Contrast with *asynchronous coordination*.

Table Lookup: Table lookup is an implementation optimization that can make many algorithms run more efficiently. A lookup table is an array that replaces runtime computation with a simple array indexing operation. The tables can be *precomputed* and stored in static storage. This pattern assumes a reasonable amount of reuse and that array lookup is faster than computing the value as needed.

Tail Recursion: A form of recursion where a result of the recursive call is returned immediately without modification to the parent function. Such forms of recursion can be rewritten to use an iteration pattern instead.

Target Processor: A specialized co-processor to which work can be offloaded. See also *host processor*.

Task: A logically discrete chunk of computational work. A task is typically a program or program-like set of instructions that is executed by a single processing core. See also *task-level parallelism* and *parallel program*.

Task-Level Parallelism: A type of *parallelism*. With task-level parallelism, also known as *function parallelism* or *control parallelism*, a programmer decomposes an algorithm into multiple sub-tasks (typically heterogeneous) and assigns each sub-task to its own parallel thread. A *DAG* is typically used to model the flow of data through the network of sub-tasks. Task-level parallelism doesn't scale well as it is difficult for programmers to decompose a program into more than just a few parallel sub-tasks.

TBS: See *Thread Block Size*.

Template Metaprogramming: The use of generic programing techniques to manipulate and optimize source code before it is compiled. The template rewriting rules in C++ can be interpreted as a functional language for manipulating C++ source code. These rules can be used to generate high performance optimized code at compile time instead of run-time. See also *metaprogramming*.

Temporal Locality: The frequent re-use of the same data element within a small period of time. See also *locality* and *spatial locality*.

Test-and-Set: test and set is an *atomic operation* used to synchronize *concurrency* issues and prevent incorrect multi-threaded behavior. It writes a new value to a memory location and then returns the old value from the same memory location. Even though this actually requires several instructions in hardware, it appears as a single atomic operation to the rest of the system, i.e. no other thread can interrupt the CAS once it has begun and until it has finished. This has been replaced by *compare-and-swap* semantics in most modern architectures. See also *fetch-and-add*.

Thread: The smallest unit of execution that can be scheduled on a single processing core. Each thread maintains thread state (registers, instruction pointer, …) for correct execution. See also *Thread Warp* and *Thread Block*.

Thread Block: A CUDA thread block of a fixed shape (1D, 2D, or 3D) and size. Each thread block is eventually scheduled to run concurrently on a SM to execute a parallel GPU kernel algorithm. For best performance, each thread block is made up of one or more *thread warps*. Although, thread blocks which are not a multiple of the *WarpSize* (=32) are supported, they tend to have poor performance as one or more SP cores are *underutilized*. Each thread block is typically assigned one or more matching *data block*s to process. See also *CTA*, *Thread Warp*, and *Data Warp*.

Thread Block Size: A fixed-size *thread block* which contains TBS elements, where TBS is typically computed as *TBS* = *nWarps*WarpSize*. See also *Data Block Size*.

Thread Constraints: Various hardware constraints that limit the size (and shape) of threads on a CPU (or GPU).

Thread Context: The state information (registers, instruction pointer, etc.) about the executable state of a thread which is needed to correctly resume execution after a thread has been paused or blocked.

Thread-Level Parallelism (TLP): A type of *parallelism*. With Thread-Level Parallelism (TLP), also known as *multi-threading*, work is sub-divided and executed on multiple independent *threads* of execution in parallel. Typically a thread scheduler maps a pool of m threads onto p cores (with $m \geq p$). TLP enables *task*- and *data-level parallelism*.

Thread Lifecycle: A state machine diagram that captures common thread states and what behavior moves a thread between the various thread states. See also *thread state*.

Thread-Local Storage: With thread-local storage, each thread has its own private copy of local data (variables, stacks, registers, program counter, etc.). These variables retain their values across sub-routine and other code boundaries are thread-safe since they uniquely belong to each individual thread. Parallel threads may end up executing the same code yet all threads still refer to their own unique local data.

Thread Manager: A thread manager is responsible for creating, tracking, managing, and destroying threads in a multi-threading system. Threads are a mechanism for running *concurrent* tasks using a single core or multiple *parallel* tasks using multiple cores. A thread manager can be implemented in hardware and/or software. See also *thread*, *multi-threading*.

Thread-Safe: A program (or section of code) is said to be thread-safe if the algorithm manages data structures in a way that guarantees correct execution by multiple threads in parallel. In other words, the multi-threaded parallel program gives the same results as an equivalent single-threaded serial program. A program can be made thread-safe by 1) *partitioning* data into *independent* per-thread *runs*, 2) supporting *re-entrancy*, *mutual exclusion*, and *thread-local* storage or 3) by using *lock-free* or *wait-free* algorithms and data structures.

Thread Scheduler: See *thread manager*.

Thread State: The various states that a *thread* can be in over the course of its lifetime from creation to termination. A *running thread* is the thread that is currently executing on the processor core. An *active* thread (or blocked thread) is one that is waiting to resume execution on the core. A *stalled thread* is one that is waiting to wake-up or for a long I/O operation on which it depends to complete. An *aborted thread* is a thread which has been requested to be terminated but is not yet terminated.

Thread Warp: The smallest unit of vectorized execution that can be scheduled on a single streaming multi-processor (SM) by being mapped onto multiple SP cores and other functional processing units. Each warp also maintains associated *warp state* for correct execution. On modern GPUs, each thread warp consists of *WarpSize* (=32) threads. See also *Data Warp* and *Thread Block*.

Throughput: A measure of performance where we count the number of units of work completed per unit of time, typically this is measured as work per second, mega-items per second, or giga-items per second. GPUs are throughput focused devices, IE, GPUs process hundreds (or thousands) of instructions in parallel using hundreds (or thousands) of simple processing cores. A throughput device (or program) might put up with higher latency for an individual computation if it still resulted in faster throughput overall. Contrast with *latency*. See also *data-*, *instruction-* and *I/O-throughput*.

Thrust: NVIDIA's thrust is a C++ template library of data-parallel algorithms and data-parallel data structures. Thrust is intended to eventually have the same kind of relationship to C++ for data-parallel algorithms on GPUs that the C++ *standard template library* has for serial algorithms on CPUs. It currently supports GPU accelerated algorithms for sort, scan, transform, and reduce.

Tile: A region of memory which is part of some larger collection. Typically each tile is 2D. Tiles might result from applying the *partition pattern* to data which is inherently 2D. See also *run* and *block*

Tiling: With tiling, also known as blocking, a loop is divided into a set of parallel tasks of a suitable *granularity*. In general, tiling consists of applying multiple steps on a smaller part of a problem instead of running each step on the whole problem one after the other. The main purpose of tiling is to increase the reuse of data in caches for better performance. For example, with parallel matrix multiplication $\boldsymbol{C} = \boldsymbol{A} \times \boldsymbol{B}$, we might divide the original $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$ matrices into a collection of 2D tiles small enough so that three subtiles $\boldsymbol{A}_{ik}, \boldsymbol{B}_{kj}$, and $\boldsymbol{C}_{ij}$ fit into cached (or shared) memory all at the same time and then run the parallel matrix multiplication in terms of these sub-tiles. Contrast the tiled access pattern with the usual row-by-row vs. column-by-column serial solution for matrix multiplication.

Time Complexity: A complexity measure for the amount of time used by an algorithm as a function of problem size. See also *space complexity* and *asymptotic analysis*.

Time Elapsed: A measure of the duration of time from the start to the end of some program, function, or section of code. See also *latency*.

Time Slice: A time slice is a short period of time (typically counted in machine cycles) for which a *thread* is allowed to execute before being *pre-empted* by the scheduler (hardware or software) in favor of another *active* thread.

TLB: A translation lookaside buffer (TLB) is a specialized cache used to hold translations of virtual to physical page addresses. The number of elements in the TLB determines how many pages of memory can be accessed simultaneously with reasonable efficiency. Accessing a page not already in the TLB will cause a *TLB miss*. A TLB miss causes a trap (system interrupt) to the operating system to load the requested page into memory and to update the TLB. The TLB enables *virtual memory management*.

TLB miss: Occurs when a virtual memory page access is made for which the page translation is not already in the TLB.

TLP: See *thread-level parallelism* or *multi-threading*.

Tomasulo's Method: A type of ILP hardware architecture which enables out-of-order execution of instructions. Tomasulo's method improves on scoreboarding by adding support for register renaming (aliasing), reservation stations (queues), a re-order buffer, and a command data bus (CDB) for broadcasting updated data values to all compute stations. Tomasulo's method reduces pipeline stalls by directly resolving false resource conflicts (WAW & WAR hazards) through registers by register renaming.

Topological Sort: A topological sort of a *directed acyclic graph* (DAG) is a linear re-ordering of the graphs vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering. Often used for imposing a valid ordering on a dependency graph, although there are frequently many different but all equally valid orderings. Topological sorts can be computed in linear time, $O(n)$.

Transaction: An atomic update to a data element. Each data element is assumed to consist of multiple fields where some or all fields might be potentially changed as part of the update. Atomic means that the results of all the individual field updates are either seen in their entirety or none of the individual field updates are seen when looking at the data element, in other words, any data changes are seen by the rest of the system as all or nothing.

Transactional Memory: A way of accessing memory such that a collection of memory updates, called a transaction will be visible to other threads (or tasks) either all or nothing. Transactional memory is an alternate approach to handling *resource competition* which is different from *mutual exclusion*.

Translation Lookaside Buffer: see *TLB.*

Transpose Pattern: A special case of a gather and scatter pattern. Where we transform 2D data stored in a row-major order into a column-major order or vice versa.

Tree-Reduce: Tree Reduce is a parallel pattern that reduces two input elements per thread to one output sum at each stage and, in so doing, takes a logarithmic $O(\log_2 n)$ number of stages to fully reduce n elements down to one final sum. Each pair-wise reduction uses an associative binary operator which has an associated *identity* element. The first stage requires $p = \lceil n/2 \rceil$ threads. Each subsequent stage uses half as many threads. Compare with *Run-Reduce*.

Trim Test: A simple 1D interval test used to eliminate non-overlapping sub-trees during a *kd-tree* search algorithm and thus speed up *nearest-neighbor searches*.

True-Dependency: See *read after write (RAW).*

Under-Utilization: A processing core is said to be underutilized when it could be doing useful work but is instead sitting idle. ILP and TLP techniques in a broad sense are trying to increase the utilization of all parallel processing cores on a machine.

Uniform Parameter: A constant parameter to the *map pattern* that is broadcast to all instances of the map's *elemental function*. See also *varying parameter*.

Unpack Pattern: The inverse of the *pack pattern*, this operation scatters data from a smaller packed array back into a larger unpacked array. This pattern may optionally fill in a default value for missing data.

Unsplit Pattern: The inverse of the *split pattern*, this operation scatters data back into its original location. Unlike the case with the unpack pattern, there is no missing data to worry about.

Unstable Sort: See *stable sort*.

Unswitching: Unswitching is a *loop optimization* where a conditional inside a loop is moved outside of the loop by duplicating the loop twice once each for the {*true*} and {*false*} clauses.

Unzip pattern: The inverse of the *zip pattern*, this operation deinterleaves data and can be used to convert from an *array-of-structures (AoS)* to a *structure-of-arrays (SoA)*.

Varying Parameter: A parameter to the *map pattern* that delivers a different parameter to each instance of the map's *elemental function*. See also *uniform parameter*.

Vector Intrinsic: An *intrinsic* used to specify a vector-parallel operation.

Vector-parallel operation: A low-level operation that can act on multiple data elements at once in SIMD fashion.

Vector-Level Parallelism: A type of *parallelism*. Vector-level Parallelism, also known as *vectorization*, applies a single instruction to an array of data in parallel using an array of processing cores. This is the heart of the SIMD parallel model.

Vectorization: Reorganizing code to support *vector-level parallelism*.

Vector Processor: Multiple data is executed in parallel using an array of 1D simple cores and corresponding data registers called a vector register. Typically all vectors move in lock-step through the same instruction stream. This is the essence of the SIMD parallel processing model.

Virtual Memory: Virtual memory decouples the address used by software from the physical address where data is stored in physical memory. The translation from virtual addresses to physical addresses is done in hardware which is managed by the operating system.

Virtual Memory Management (VMM): A memory management technique developed for multi-tasking operating systems which decouples logical memory addresses from physical memory address. The O.S. with hardware support translates the logical addresses into physical addresses at runtime. This technique hides the complexity of actual *memory hierarchies* from programmers so they can continue to write programs using the much simpler *RAM memory model*.

VLIW (Very Large Instruction Word): A processor architecture that explicitly supports processing multiple instructions in parallel which are stored in a single VLIW. See *Vectorization*.

Von Neumann Model: An abstract model of a serial computer architecture for programmers. Under this model, a computer consists of a CPU, a load/store memory and a bus to transfer data between the CPU and memory. The memory stores both programs (as sequential streams of instructions) and data (both input and output). The CPU fetches instructions sequential in-order from the program, decodes, and executes the instructions. Data is transferred between CPU registers and memory using simple load/store commands. The CPU performs all the basic computations. I/O is used to get data (and programs) into the computer memory from other computers, devices, or human operators. See also the *RAM machine model*, the *PRAM machine model*, and the *multi-computer machine model*.

Voronoi Diagram: A voronoi diagram is a *spatial partitioning data structure* used to speed up geometric searches. The set of generating points (called seeds) is specified a-priori, and for each seed point there is a corresponding geometric region (typically as a polygon) consisting of all points closer to that seed point than to any other seed point. See also *BSP tree*, *octree*, and *kd-tree*.

Voting Intrinsics: Special voting instructions on a GPU that allow all threads in a warp to share per-thread predicate test results with all other threads in a warp in a single instruction. These Voting instructions are also used internally by the GPU hardware to implement *barriers*.

Wait-free algorithm: A *non-blocking algorithm* is also wait-free if there is guaranteed per-thread progress. See also *lock-free algorithm*.

Warp: The smallest unit of vectorized execution that can be scheduled on a single streaming multi-processor (SM) by being mapped onto the SP cores and other functional processing units. Each warp also maintains associated *warp state* for correct execution. See also *Data Warp* and *Thread Warp*.

Warp-by-Warp BASk: An efficient access pattern at the bottom level of the 2-level CTA mapping that supports coalescence. Each thread warp of size *WarpSize* within the thread block is assigned its own unique chunk of data within the data block of size (*nWork***WarpSize*). Each thread within the thread warp accesses its own unique data element within the current group of *WarpSize* elements and then strides (*stride*

= *WarpSize*) to the next group of *WarpSize* data elements within the data chunk to access. This approach is more coherent and requires less barrier synchronization than the *Block-By-Block* BASk. Most importantly, each individual thread warp has *Warp Independence* and thus can proceed to work on its own assigned chunk of data without concern to other concurrent thread warps doing the same. See also *block access skeleton*. Compare with the *Block-by-Block BASk*.

Warp Context: The state information (instruction pointer, etc.) about the executable state of a warp which is needed to correctly resume execution after a warp has been paused.

Warp Independence: With warp independence, each individual *thread warp* within a *thread block* can proceed to work on its own assigned data chunk independent of any other concurrently executing thread warp. See also *Warp-by-Warp BASk*.

Warp Lifecycle: A state machine diagram that captures common warp states and what behavior moves a warp between the various warp states.

Warp Padding: With warp padding, the *pad and rake* technique is used to mitigate *bank conflicts* for runs that are a power of two {2, 4, 8, …} in length. It works by padding a single unused pad column after (or before) each data warp (effectively turning an even run of 32 elements into an odd run of 33 elements). Indexing must be raked to skip over the pad columns. See also *run padding*.

Warp Reduce: A building block for GPU Reduce/Scan that cooperatively reduces n = [2,4,8,16,32] elements down to a total sum using n threads in parallel. The reduction is done using a binary associative summation operator. See also *Serial Reduce*, *Serial Scan*, and *Warp Scan*.

Warp Scan: A building block for GPU Reduce/Scan that cooperatively scans n = [2,4,8,16,32] elements into a prefix sum using n threads in parallel. The scan is done using a binary associative summation operator. See also *Serial Reduce*, *Serial Scan*, *Warp Reduce*, and *Run Update*.

Warp State: The various states that a *warp* can be in over the course of its lifetime from creation to termination. An *executing warp* is the warp that is currently running on the SP cores that make up an SM(X) core. An *active warp* is a warp that could be running but is currently waiting to be scheduled onto the SP cores. An *inactive warp* is a warp that is currently stalled on a short-term instruction dependency or waiting on a long-term I/O operation to complete before it is again ready to resume execution.

***WarpSize*:** Two related definitions: 1) A group of *k*=32 threads that collectively execute the same instruction stream in lockstep, but on different data elements. Modern GPU architectures are designed to support SIMD efficiently via *data-level parallelism* using *thread block*s made of up of *thread warp*s where each thread warp is of size 32. 2) The *WarpSize* parameter represents *Data-level parallelism* on SIMD hardware. This parameter allows experiments with data-level parallelism to find the best performance. The programmer must use generic programming techniques to support parallel processing one thread warp at a time. Unlike the *nWork* and *nWarps* parameters, this parameter is typically fixed at *k*=32, as this is what modern GPUs efficiently support.

Weak Scalability: A form of scalability that measures how performance increases when adding in additional parallel computing resources while increasing the problem size at the same rate.

Weakly Parallel Algorithm: With *Weakly Parallel algorithms*, the *depth efficiency* is approximately $D(n) \approx O\left(\frac{n}{\log n}\right)$. The algorithm usually has significant section(s) of the code which cannot be parallelized easily so *Amdahl's Law* bounds performance. Alternately, the weakly parallel algorithm requires so much coordination (communication) between parallel cores that the resulting *parallel overhead* overwhelms most of the advantage of using parallel processing cores.

Work: Three definitions: 1) the part of a parallel program which actually spends time solving the original problem for which the program was written instead of time spent coordinating (or communicating) with other threads (tasks). See also *parallel overhead*. 2) A measure of how many data elements get processed per-thread on each function (or kernel) invocation. 3) An abstract unit of useful computations. See also *work-depth analysis*.

Work Complexity: The asymptotic total number of operations, $W(n,p)$, required by a parallel algorithm to run across all parallel threads (p) as a function of problem size (n). Work complexity is essentially equivalent to *asymptotic analysis* for serial programs (p=1). If the work complexity of a parallel algorithm is asymptotically the same as an equivalent serial algorithm (i.e. both require linear work, $O(n)$) then the parallel algorithm is said to be work-efficient. If the parallel algorithm is asymptotically more expensive (log-linear, $O(n \log n)$ vs. linear work $O(n)$), then the parallel algorithm is said to be work-inefficient. In order for speedup ratios to be computed, it is often better to use *Big "Theta" notation* instead of *Big "O" notation*. See also *depth complexity*.

Work+Depth Analysis: A model for analyzing the performance of parallel programs that can be used to compute both upper and lower bounds on parallel *speedup*. See also *depth complexity* and *work complexity*.

Work-Span: See *Work+Depth analysis*.

Workpile Pattern: An extension to the *map pattern* that allows new work items to be dynamically generated and added to the workpile during the execution of each *elemental function*. If the map pattern can be thought of as a parallel generalization of the **`for`** statement then the workpile pattern can be thought of as a parallel generalization of the **`while`** statement.

Work Stealing: A scheduling model where each thread (or task) has it's own local work queue. The initial work load is load balanced across all threads and stored in each threads local queue. As new dynamic work gets generated by each thread (or task), each thread adds it onto its own local queue. What a specific thread runs out of work in its own queue, it will randomly steal work from another thread's work queue. When all local queues are empty, then the task is typically complete. This model greatly reduces resource competition for the work queue data structures themselves which reduces parallel overhead. Load balancing though not perfect tends to be quite well across all threads.

Wrap-around Arithmetic: Computers usually use fixed size data types which have a specific numeric precision and range. When computers perform simple computations such as addition or multiplication there needs to be a solution when the resulting computed value is logically outside the allowed for that data type. One solution is simply to allow the values to wrap around naturally to the other end of the range. For instance, with unsigned 8-bit integers with a range [0..255]. Adding 250 + 20 = 270 which is outside the allowed range, with wrap-around arithmetic the actual stored result would become 15 = 270 % 255. Contrast with *saturation arithmetic*.

Write after Read (WAR): A type of *data hazard*. With WAR, one instruction tries to write to a destination (register, memory) before a prior instruction has had a chance to read from the same destination. Unless this hazard is prevented the read instruction may pick up an incorrect value. A WAR dependency can be handled by *stalling* or *register renaming*.

Write after Write (WAW): A type of *data hazard*. Where one instruction tries to write to a destination (register, memory) before a prior instruction has had a chance to write to the same destination. The WAW data hazard will only cause problems if writing to the destination generates side-effects or if some parallel processing unit actually depends on the first write value (IE is reading). A WAR dependency can be handled by *stalling* or *register renaming*.

Zip Pattern: A special case of the *gather pattern* that interleaves elements from multiple arrays (or collections). This can be used to convert from a *structure-of-arrays (SoA)* to an *array-of-structures (AoS)*.

BIBLIOGRAPHY

[1] Alexandrescu, A., 2001, *Modern C++ Design, Generic Programming and Design Patterns Applied*, Addison-Wesley, Boston, MA

[2] Amdahl, G.M., 1967, Validity of the single-processor approach to achieving large scale computing capabilities, *AFIPS Conference Proceedings*, vol 30, AFIPS Press, Reston, VA, pp. 483-485

[3] Arya, S., and Mount., M., 1993, Algorithms for Fast Vector Quantization, IEEE *Proceedings of Data Compression Conference*, IEEE Computer Society Press, pp. 381-390.

[4] Atallah, M., and Blanton M., editors, 2010, *Algorithms and Theory of Computation Handbook,* 2nd Edition, Volume 2, *Special Topics and Techniques*, Taylor and Francis Group LLC, Boca Raton, FL

[5] Bell, N., and Hoberock, J., 2012, Thrust: A Productivity-Oriented Library for CUDA, in W.M. Hwu, editor, *GPU Computing Gems: Jade Edition*, chapter 26, pp. 359-371, Elsevier Inc., Waltham MA, 2012. URL: https://code.google.com/p/thrust/downloads/detail?name=An%20Introduction%20To%20Thrust.pdf

[6] Bentley, J. L., 1975, Multidimensional binary search trees used for associative searching, *Communications of the ACM*, Sept. 1975, Vol 18(9), pp. 509-517 URL: http://dl.acm.org/citation.cfm?id=361007

[7] de Berg, M. et al, 2000, *Computational Geometry, Algorithms and Applications*, 2nd Edition, Springer Verlag, New York NY

[8] Blelloch, G., 1989, Scans as Primitive Parallel Operations. *IEEE Transactions on Computers* 38(11), pp. 1526-1538

[9] Blelloch, G., 1990, Prefix Sums and Their Applications, School of Computer Science, Carnegie Mellon University, CMU-CS-90-190, Nov. 1990. URL: http://www.cs.cmu.edu/~scandal/papers/CMU-CS-90-190.html

[10] Blelloch, G., et al, 1995, Solving Linear Recurrences with Loop Raking, *Journal of Parallel and Distributed Computing*, Academic Press Inc., vol. 25, Iss. 1, pp 91-97.

[11] Blelloch, G. and Maggs, B., 1996, Parallel Algorithms. ACM Comput. Surv. v. 28, 1 (March 1996) p. 51-54

[12] Brent, R. and Kung, H., 1982, A Regular Layout for Parallel Adders, IEEE Transactions on Computers, vol. C-31, no 3, p. 260-264, March 1982

[13] Brown, S. and Snoeyink, J., 2012, Modestly Faster Histogram Computations on GPUs, In IEEE *Innovative Parallel Computing* (INPAR '12), May 2012

[14] Buck, I, et al, 2004, Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Transactions on Graphics*, 23(3), pp. 777-786, 2004. URL: http://dl.acm.org/citation.cfm?doid=1186562.1015800

[15] Bustos, B., Deussen, O., Hiller, S., Keim, D., 2006, A Graphics Hardware Accelerated Algorithm for Nearest Neighbor Search, *Proceedings of the 6th International Conference on Computational Science*, May 2006, pp. 196-199

[16] Chatterjee, S., et al, 1990, Scan Primitives for Vector Computers, in *Supercomputing '90, Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, pp. 666-675, 1990. URL: http://dl.acm.org/citation.cfm?id=110597

[17] Cormen, T., Leiserson, C., Rivest, R., and Stein, C., 2009, *Introduction to Algorithms*, 3rd Edition, Massachusetts Institute of Technology, Boston, MA

[18] Cuccuru G., Gobbetti E., Marton F., Pajarola R., Pintus R., 2009, Fast low-memory streaming MLS reconstruction of point-sampled surfaces, *Graphics Interface,* pp. 15-22.

[19] Dongarra J., et al, 2003 *Sourcebook of Parallel Computing*, Morgan-Kaufman, San Francisco CA

[20] Dotsenko, Y., et al, 2008, Fast Scan Algorithms on Graphics Processors, in ICS'08, Proceedings of the 22nd annual international conference on Supercomputing, pp. 205-213, 2008.

[21] Edmonds, J., 2008, *How to think about Algorithms*, Cambridge University Press, New York, NY

[22] Flynn, M. J., 1972, Some Computer Organizations and Their Effectiveness, in IEEE Trans. Comput. C-21 (9): 948-960. doi:10.1109/TC.1972.5009071

[23] Garcia, V., Debreuve, E., and Barlaud, M., 2008, Fast k Nearest Neighbor Search using GPU. *Proceedings of Computer Vision and Pattern Recognition Workshops*, June 2008, pp. 1-6.

[24] Garland, M., Kirk, D., 2010, Understanding Throughput-Oriented Architectures, *Communications of the ACM* 31(1), pp. 58-66, Nov, 2010.

[25] Göddeke, D., Kurzak, J., Weiß, J-P., Heidekrüger, A., and Schröder, T., Scientific Computing on GPUs: GPU Architecture Overview, *PPAM 2011 Tutorial*, Toruń, Poland, Sept. 11, 2011, URL: http://gpgpu.org/wp/wp-content/uploads/2011/09/03-arch.pdf

[26] Govindaraju, N.K., et al, 2005, GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management, *Microsoft Technical Report* MSR TR-2005-183, Nov. 2005, revised March 2006. URL: http://research.microsoft.com/pubs/64572/tr-2005-183.pdf

[27] Gustafson, J. L., 1988, Reevaluating Amdahl's Law, *Communications of the ACM* 31(50), pp. 532-533 URL: http://www.johngustafson.net/pubs/pub13/amdahl.pdf

[28] Gustafson, J. L., 2011, Brent's Theorem, Encyclopedia of Parallel Computing, pp. 182-185 URL: http://link.springer.com/referenceworkentry/10.1007/978-0-387-09766-4_80

[29] Jensen H., 2001, *Realistic Image Synthesis Using Photon Mapping*, A K Peters, Natick MA

[30] Ha, L., Krüger, J., and Silva, C., 2009, Fast 4-way Parallel Radix Sorting on GPUs, *Computer Graphics Forum* V28, pp. 2368-2378, Blackwell Publishing. URL: http://onlinelibrary.wiley.com/doi/10.1111/j.1467-8659.2009.01542.x/abstract

[31] Han, T. and Carlson, D., 1987 Fast Area-Efficient VLSI Adders, Proc. 8th Symp. Comp. Arith. P 49-56, Sept. 1987

[32] Harris, D. and Sutherland, I., 2003, Logical Effort of Carry Propagate Adders, Asilomar Conference on Single, Systems, and Computers. URL: http://www3.hmc.edu/~harris/research/adderle.pdf

[33] Harris, M., 2007, *Optimizing Parallel Reduction in CUDA*,
URL: http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

[34] Harris, M. , Shubhabrata S., and Owens J.D. , 2008, Parallel Prefix Sum (Scan) with CUDA., in *GPU Gems 3*, edited by H. Nguyen, chapter 39 pp. 851-876 . Addison-Wesley, Reading MA URL: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html

[35] Hennessey, J. and Patterson, D., 2012, *Computer Architecture: A Quantitative Approach*, 5th Edition, Elsevier Inc, Waltham, MA

[36] Hillis, W. Daniel and Guy L. Steele, Jr. 1986, Data Parallel Algorithms, *Communications of the ACM* 29(12), pp. 1170-1183

[37] Hollerith, H., 1889, Art of Compiling Statistics, U.S. Patent 395,781, patented Jan.8, 1889

[38] Hwu, WM, editor, 2011, *GPU Computing Gems Emerald Edition*, Elsevier Inc., Waltham MA

[39] Hwu, WM, editor, 2012, *GPU Computing Gems Jade Edition*, Elsevier Inc., Waltham MA

[40] Knowles, S., 1999, A Family of Adders, Proc. 14th IEEE Symposium on Comp. Arith., p. 30, April, 1999

[41] Knuth, D., 1998, *The Art of Computer Programming*, Volume 3, *Sorting and Searching*, 2nd Edition, Addison-Wesley, Boston, MA

[42] Kogge, P., and Stone, H., 1973, A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations, IEEE Transactions on Computers, C-22, p. 783-791

[43] Khronos Group, 2012, *OpenCL – The open standard for parallel programming of heterogeneous systems*, URL: www.khronos.org/opencl/

[44] Ladner, R. and Fischer, M., 1980, Parallel Prefix Computation, Journal of the ACM (4): p. 831-838

[45] Merrill, D. and Grimshaw, A., 2010, *Parallel Scan for Stream Architectures.*, Technical Report, CS2010-02, University of Virginia, Dept. of Computer Science, Charlottesville, VA URL: http://www.cs.virginia.edu/~dgm4d/papers/ParallelScanForStreamArchitecturesTR.pdf

[46] Merrill, D. and Grimshaw, A., 2010, *Revisiting Sorting for GPGPU Stream Architectures.*, Technical Report CS2010-03, University of Virginia, Dept. of Computer Science, Charlottesville, VA URL: http://www.cs.virginia.edu/~dgm4d/papers/RadixSortTR.pdf

[47] Micikevicius, P., 2010, Analysis-Driven Optimization, *NVIDIA GPU Technology Conference 2010*, URL: http://www.NVIDIA.com/content/GTC-2010/pdfs/2012_GTC2010v2.pdf

[48] Micikevicius, P., 2010, Fundamental Optimizations, *NVIDIA GPU Technology Conference 2010*, URL: http://www.NVIDIA.com/content/GTC-2010/pdfs/2011_GTC2010.pdf

[49] Microsoft, 2012, *C++ AMP Overview*, URL: http://msdn.microsoft.com/en-us/library/hh265136.aspx

[50] Miller, R., and Boxer, L., 2013, *Algorithms Sequential & Parallel a Unified Approach*, 3rd Edition, CENGAGE Learning., Boston MA

[51] Mitzenmacher, M., 1996, *The Power of Two Choices in Randomized Load Balancing*, PhD Thesis, Computer Science Department, University of California at Berkeley, 1996.

[52] Mount, D. and Arya, S., 2010, ANN: A Library for Approximate Nearest Neighbor Searching, URL: http://www.cs.umd.edu/~mount/ANN/ ANN Version 1.1.2 (Release date: Jan 27,2010)

[53] Nickolls, J., and Dally, W.J., 2010, The GPU Computing Era, *IEEE Micro* 30(2), pp. 56-69, Mar-Apr, 2010.

[54] Nugteren, C., van den Braak, G.J., Corporaal, H., and Mesman, B., 2011. High performance predictable histogramming on GPUs: exploring and evaluating algorithm trade-offs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units* (GPGPU-4). ACM, New York, NY, USA, , Article 1 , 8 pages. DOI=10.1145/1964179.1964181 URL: http://doi.acm.org/10.1145/1964179.1964181

[55] NVIDIA, 2010, *What is CUDA*, URL: http://www.NVIDIA.com/object/what_is_cuda_new.html

[56] NVIDIA, 2012, *CUDA C Best Practices Guide*, URL: http://docs.NVIDIA.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf

[57] NVIDIA, 2012, *CUDA C Programming Guide*, URL: http://docs.NVIDIA.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[58] NVIDIA, 2012, *NVIDIA GeForce GTX 680*, Whitepaper, URL: http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf

[59] NVIDIA, 2010, *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, Whitepaper, URL: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[60] NVIDIA, 2012, *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*, Whitepaper, URL: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[61] NVIDIA, 2012, *Parallel Thread Execution ISA Version 3.1*, URL: http://docs.NVIDIA.com/cuda/pdf/ptx_isa_3.1.pdf

[62] NVIDIA, 2012, Thrust, URL: https://developer.NVIDIA.com/thrust

[63] Owens, J., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, A.E, and Purcell, T.J, , 2007, A Survey of General-Purpose Computation on Graphics Hardware, in *Computer Graphics Forum* 2007 26(1), pp.80-113, Mar. 2007.

[64] Pajarola, R., 2005, Stream-Processing Points. *IEEE Visualization*, p. 31.

[65] Patterson, D.A., 2009, The Top Innovations in the New NVIDIA Fermi Architecture, and the Top 3 Next Challenges, Elec. Eng. and Comp. Sci., UC Berkeley, Sept. 2009. URL: http://origin-jp.nvidia.com/content/PDF/fermi_white_papers/D.Patterson_Top10InnovationsInNVIDIAFermi.pdf

[66] Pearson, K., 1895, Contributions to the Mathematical Theory of Evolution II. Skew Variation in Homogenous Material, *Philosophical Transactions of the Royal Society A: Mathematical, Physical, and Engineering Sciences* v. 186 pp. 343-326

[67] Podlozhnyuk, V., 2007, *Histogram calculation in CUDA*, Technical Report, NVIDIA, URL: http://developer.download.NVIDIA.com/compute/cuda/1_1/Website/projects/histogram256/doc/histogram.pdf

[68] Press, W., Teukolsky, S., Vetterling, W., and Flannery, B., 2007, *Numerical Recipes the Art of Scientific Computing*, 3rd Edition, Cambridge University Press, New York, NY

[69] Purcell, T.J, Donner, C, Cammarano, M., Jensen, H.W., and Hanrahan, P., 2003, Photon Mapping on Programmable Graphics Hardware. *ACM SIGGRAPH / EUROGRAPHICS Conference on Graphics Hardware*, ACM Press

[70] Qiu, D., May, S., and Nuchter, A., 2009, GPU-accelerated Nearest Neighbor Search for 3D Registration, *7th International Conference on Computer Vision Systems*, Oct 2009, pp. 194-203

[71] Raab, M., and Steger, A., 1998, "Balls into Bins" – A Simple and Tight Analysis, *Conference on Randomization and Approximation Techniques in Computer Science* (*RANDOM*), pp. 159-170, 1998.

[72] Rauber, T. and Rünger, G., 2010, *Parallel Programming for Multicore and Cluster Systems*, Springer-Verlag, Berlin Heidelberg

[73] Rozen, T., Boryczko, Alda, W., 2008, GPU Bucket Sort Algorithm with Applications to Nearest-Neighbor Search, *Journal of the 16th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, Feb. 2008

[74] Sanders J., and Kandrot, E., 2011, *CUDA by Example: An introduction to General-Purpose GPU programming*, Elsevier Inc., Waltham MA

[75] Samet, H., 2006 *Foundations of Multidimensional and Metric Data Structures*, Morgan Kaufmann, San Francisco CA

[76] Sedgewick, R., 1998, *Algorithms in C++, 3rd Edition, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching*, chapter 7 pp. 315-346, Addison-Wesley, Reading MA

[77] Shakhnarovich, G., Darrell, T, and Indyk P., Editors, 2005, *Nearest-Neighbor Methods in Learning and Vision, Theory and Practice*, The MIT Press, Cambridge MA

[78] Skiena, S. S., 2008, *The Algorithm Design Manual, 2nd Edition*, Springer-Verlag, London UK

[79] Shams, R. and Kennedy, R., 2007, Efficient Histogram Algorithms for NVIDIA CUDA compatible devices, in *ICSPCS: Proceedings of the International Conference on Signal Processing and Communication Systems*

[80] Sklansky, J., 1960, *Conditional-Sum Addition Logic*, IEEE Transactions on Electronic Computers, vol. EC-9, no. 2, p. 226-231

[81] Stroustrup, B., 2013, *The C++ Programming Language, fourth edition*, Pearson Education, 2013, Upper Saddle River, NJ

[82] Vaidya, P. M., 1989, An O(n log n) Algorithm for the All-Nearest Neighbors Problem., *Discrete and Computational Geometry*, (4):101-115, 1989

[83] Volkov, V., 2010, Better Performance at Lower Occupancy, *NVIDIA GPU Technology Conference 2010*, URL: http://www.NVIDIA.com/content/GTC-2010/pdfs/2238_GTC2010.pdf

[84] von Neumann, J., 1945, First Draft of a Report on the EDVAC, archived from the original on March 14, 2013, retrieved August 24, 2011 URL: https://web.archive.org/web/20130314123032/http://qss.stanford.edu/~godfrey/vonNeumann/vnedvac.pdf

[85] Wadleigh, K.R., and Crawford, I. L., 2000, *Software Optimization for High Performance Computing*, Prentice Hall PTR, Upper Saddle River, NJ

[86] Wilt, N., 2013, The CUDA Handbook: *A Comprehensive Guide to GPU Programming*, Addison-Wesley, Upper Saddle River, NJ

[87] Yang, Z., Zhu, Y., and Pu, Y., 2008, Parallel Image Processing Based on CUDA, In *Computer Science and Software Engineering, 2008 International Conference on*, v3, pp. 198-201

[88] Zhou, K., Hou, Q., Wang, R., and Guo B., 2008, Real-time KD-Tree Construction on Graphics Hardware, *ACM SIGGRAPH Asia 2008*, April 2008, page 10.