PARALLEL CACHE-EFFICIENT ALGORITHMS ON GPUS

A DISSERTATION SUBMITTED TO THE GRADUATE DIVISION OF THE UNIVERSITY OF HAWAIʻI AT MĀNOA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMPUTER SCIENCE

AUGUST 2023

By

Kyle M. Berney

Dissertation Committee:

Nodari Sitchinava, Chairperson
Henri Casanova
Peter Sadowski
Daniel Suthers
June Zhang

Keywords: parallel algorithms, GPU, cache-efficent, bank conflicts, models of computation

ABSTRACT

Graphics Processing Units (GPUs) have emerged as a highly attractive architecture for general-purpose computing due to their numerous programmable cores, low-latency memory units, and efficient thread context switching capabilities. However, theoretical research on parallel algorithms for GPUs is challenging due to the multitude of interdependent factors influencing overall runtime. Computational models are commonly employed to provide simplified abstractions of computing system architectures. However, developing a computational model that is both simple and accurate, encompassing all performance-affecting aspects of GPU algorithms, is a seemingly impossible task. Existing GPU models often incorporate numerous variables to account for specific performance factors, rendering them less accessible to researchers.

This dissertation obviates the lack of a widely accepted model of computation for GPUs by instead employing multiple classical parallel models to capture both parallel computational complexity and cache-efficiency. Namely, we leverage existing knowledge and algorithmic techniques from the Parallel Random Access Machine (PRAM), Parallel External Memory (PEM), and Distributed Memory Machine (DMM) models to aid in the design and analysis of GPU algorithms at various levels of detail. We validate and demonstrate our approach through case studies on specific problems (e.g., sorting, searching, and single source shortest paths), providing both theoretical analysis and corresponding empirical results. Our results highlights the applicability of the selected parallel models of computation to GPUs and illustrates how theoretical research can expose valuable insights into the performance of GPU algorithms in practice.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

Within the past decade, graphics processing units (GPUs) have become an increasingly popular and powerful numerical coprocessor. Modern day GPUs provide thousands of physical cores, lightweight context switching between virtual threads, and low-latency memory units. However, due to the complexity of the GPU architecture, leveraging the full computational power of GPUs is a challenging task. In general, a high performance GPU algorithm must be both highly parallel and cache-efficient. In addition, various interdependent factors must be considered such as: the hierarchical organization of threads, the allocation of hardware resources (e.g., the number of registers used per thread), and the maximum number of active threads scheduled onto the hardware (known as *occupancy*).

Across various scientific disciplines, models are used to facilitate the understanding of a particular system and/or phenomena. For traditional parallel central processing unit (CPU) systems, numerous parallel computational models have been developed that provide a simplified, but relatively accurate, abstract view of a modern multi-core CPU architecture. These models allow theoretical researchers to design and analyze provably efficient parallel algorithms that can then be implemented and optimized on CPU systems by experimental researchers. In contrast, while several computational models for GPUs have been developed, none have been widely used yet. One reason for this is the complexity of the GPU architecture, which makes it a difficult task to provide a simple and accurate abstract model of the GPU. This generally leads to GPU models that contain a plethora of variables to account for the various factors that can impact the overall runtime. Without a widely accepted model of computation for GPUs, there currently does not exist a standard approach on how to theoretically develop and analyze algorithms for GPUs.

Rather than utilizing a single GPU model that aims to provide a single overall runtime, in this dissertation, the approach is to use several parallel models to provide a set of complexity measures that captures both parallel computational complexity and parallel cache-efficiency. This approach has the additional advantage of leveraging past knowledge and algorithmic techniques that have already been developed in these parallel models. Furthermore, if an overall runtime measure is needed, the resulting set of complexity measures can generally be reused in more accurate and elaborate GPU models (either algorithmic or performance models). For example, I/O-complexity metrics are typically considered with other factors such as bandwidth and latency to the specific memory unit.

The remainder of this chapter is organized as follows: in Section 1.1 we provide an overview of the GPU; in Section 1.2 we review models of computation developed for GPUs; in Section 1.3 we review the parallel models of computation that we adpot for the GPU; and in Section 1.4 we present an overview of the problems considered in this dissertation.

1.1 GPU Overview

CUDA (Compute Unified Device Architecture) [85, 86] is a parallel software platform that allows NVIDIA GPUs to be programmed for general purpose computing (e.g., using high-level languages such as C++). As each new generation of NVIDIA GPUs feature architectural improvements over previous generations, a *compute capability* number is assigned to each GPU model that identifies specific architectural features that the particular GPU has. We say a *compute capability* number $x.y$, has a *major number* of x and a *minor number* of y. As a rule, significant architectural changes results in an increase in the major number; and smaller architectural changes results in an increase in the minor number. Thus, newer GPU models always have a compute capability number greater than or equal to older GPU models.

From a high-level perspective, a GPU program is composed of a sequence of special procedures that execute on a GPU, called *kernels*. Each kernel is executed in parallel by a set of virtual threads, called a *grid*. A grid is organized into equal sized groups of threads, called *thread blocks*; and threads in a thread block are partitioned into groups of w threads, called *warps*. (For NVIDIA GPUs, $w = 32$.) Similarly, a GPU consists of hardware units called *streaming multiprocessors* (SMs), where each SM contains a set of programmable cores. These cores are additionally divided into partitions, called *SM processing blocks*, where each partition has a corresponding warp scheduler. Therefore when a grid is launched onto the GPU, the thread blocks of the grid are distributed to available SMs; and warps within each thread block are distributed to SM processing blocks.

All SMs on a GPU are connected to a large and slow memory space, called *global memory*; a hardware managed L2 cache; and two read-only memory spaces, *constant memory* and *texture memory*. All cores on a SM are connected to a fast and small memory space, called *shared memory*; a hardware managed L1 cache; and two read-only hardware managed caches for accesses to constant and texture memory, called *constant cache* and *texture cache* respectively. Lastly, each SM processing block contains a very fast and small *register memory* space, a warp scheduler, a dispatch unit, and a hardware managed L0 instruction cache.

Global memory: If threads in a warp concurrently access contiguous global memory locations, then these accesses can be *coalesced* together into as few global memory transactions as possible (depending on the cache-line size and size of data elements accessed). In contrast, accessing global memory locations that are strided by w locations will maximize the number of global memory transactions. For example, for 4-byte data elements, performing coalesced global memory access results in $O(1)$ number of global memory transactions compared to $O(w)$ global memory transactions for uncoalesced global memory accesses.

Constant and Texture memory: Constant and texture memory are both read-only memory units that have corresponding caches within each SM, however, they differ in their optimal access

patterns. Texture memory is optimized for 2D spatial locality, i.e., if we lay out the memory space as a 2D array then to achieve peak bandwidth we want threads in a warp to concurrently access "nearby" elements. For constant memory, peak bandwidth is achieved when threads in a warp concurrently access the same memory location. If distinct memory locations are accessed, then the requests are serialized.

Shared memory: Shared memory is organized into w *memory modules*, also known as *memory banks*. Each 128-byte aligned contiguous memory segment is distributed across $w = 32$ memory banks, so that each bank holds a 4-byte word. (For GPUs with compute capability strictly less than 5.0, memory banks can be configured to hold 8-byte words from a 256-byte aligned contiguous memory segment.) If all active threads in a warp concurrently access shared memory addresses that map to distinct memory banks, then each bank can serve the requests in parallel, known as *bank conflict free* access. However, if there exists a bank with k memory requests, then a *k-way bank conflict* occurs and the bank must serve the k memory requests in a serial manner. One exception is that threads are allowed to access the same shared memory location in the same memory bank without causing a bank conflict.

Local memory: In certain situations, the compiler may store data, whose scope is local to an individual thread, in *local memory* instead of register space. In particular, local memory will be used if there is insufficient space in registers or for arrays/data structures that the compiler determines may be accessed dynamically. It is important to note that the cost of accessing local memory is comparable to the cost of accessing global memory.

L1 and L2 cache: For GPU models with compute capability strictly less than 6.0, by default the L1 cache is used to cache accesses to local memory and the L2 cache is used to cache accesses to global memory. Using a compiler flag or inline PTX assembly code, the caching policy can be modified to additionally allow the caching of global memory accesses in L1 cache. (The compiler flag will change the caching policy for all accesses in the compiled program, while inline PTX assembly code allows for the caching policy to change on a per access basis.) However, changing the caching policy will also result in a change to the cache-line size to global memory. If only the L2 cache is used to cache accesses to global memory, then the cache-line size used is 32 bytes. On the other hand, if both the L2 and L1 cache are used to cache accesses to global memory, then the cache-line size used is increased to 128 bytes. For GPUs with compute capability 6.0 and higher, the default policy is to have global memory cached in both L1 and L2 cache; and a compiler flag or inline PTX assembly code can be used to change the policy to global memory cached in only L2 cache. Regardless of the cache setting used, the cache-line size to global memory is 32 bytes for these GPUs.

Synchronization primitives: A synchronization primitive `syncthreads` is provided to allow threads in the same thread block to synchronize with each other. For compute capability 7.0 and above, intra-warp synchronizations need to be additionally managed via a `syncwarp` primitive.

Warp scheduler: Each warp scheduler handles a static set of warps, where in each instruction time step, a warp scheduler can issue a single instruction for one of its active warps. Therefore, threads within a warp executing the same instruction are executed concurrently; this behavior is called *Single-Instruction Multiple-Threads* (SIMT). Prior to compute capability 7.0, if threads within a warp diverge in code (e.g., from a conditional statement), then each branch is executed serially. Hence, the execution of threads could be correctly viewed as *Single-Instruction Multiple-Data* (SIMD), i.e., lockstep execution of threads. Starting with compute capability 7.0, the warp scheduler maintains a program counter for each thread in a warp (rather than a single program counter for all threads in a warp). This allows for greater flexibility with branch divergence in code, as branches are no longer executed in a serial manner. Due to this, the behavior of threads is no longer restricted to lockstep execution.

Latency hiding: In order to keep the GPU hardware as busy as possible (i.e., minimize the amount of time that processors on the GPU are idle), GPUs are able to perform fast context switching between warps scheduled on the SM processing block. This effectively allows for the pipelining of memory accesses, either through having a sufficient number of warps available to be scheduled onto a SM processing block or by utilizing instruction-level parallelism (ILP), i.e., pipelining independent instructions by a single thread. The ratio of the number of active warps to the maximum number of possible active warps (hardware imposed), called *occupancy*, is a typical metric that is used to measure how well an algorithm allows the GPU hardware to hide latency via context switching. Achieving high occupancy is not as simple as launching a large number of threads, as each thread may require additional hardware resources (e.g., register or shared memory space) which are limited.

Warp-communication intrinsics: Starting with the introduction of the Kepler architecture (compute capability 3.0) in 2012, NVIDIA GPUs include warp-communication intrinsics that allow for data transfer and communication between threads in a warp.

- `shuffle`($variable, i$) reads the value of *variable* from the i-th threads register space. Requires compute capability 3.0 and higher.
- `vote`(*predicate*) returns a bitstring of length w, where the i-th bit is set to 1 if the i-th thread in the warp evaluates the *predicate* to be true. Requires compute capability 3.0 and higher.
  - `vote_all`(*predicate*) returns true if all threads evaluates *predicate* to be true; otherwise returns false (i.e., reduction-and-broadcast).

  - `vote_any`(*predicate*) returns true if any thread evaluates *predicate* to be true; otherwise returns false (i.e., reduction-and-broadcast).

- `match`(*variable*) returns a bitstring of length w, where the i-th bit is set to 1 if the *variable* stored in the i-th threads register space is equal to the current threads value of *variable*; and 0 otherwise (i.e., broadcast-and-compare). Requires compute capability 7.0 and higher.

- `reduce`(*variable*, *destination*) performs a warp wide reduction operation (add, min, max, or, xor) on *variable* for each thread in the warp and stores the result in *destination*. Requires compute capability 8.0 and higher.

Additionally, certain bit-level primitives are provided:

- `brev`(x) reverses the bit order of x.

- `clz`(x) returns the number of consecutive high-order bits set to 0 in x.

- `ffs`(x) returns the index of the first least-significant bit set to 1 in x.

- `popc`(x) returns the number of bits set to 1 in x.

1.2 GPU Models

In this section, we provide a literature review of computational models designed directly for GPUs. Note that we present only computational models (i.e., algorithmic models), rather than performance models that rely on benchmark suites to calibrate variables (e.g., the number of clock cycles to perform an operation or memory access).

1.2.1 Discrete Memory Machine (DMM) and Unified Memory Machine (UMM)

The Discrete Memory Machine (DMM) and Unified Memory Machine (UMM) models focus on memory access to shared memory and global memory, respectively [82]. In both models, p threads are connected to w memory banks through a common memory management unit (MMU). Additionally, each thread has access to r local registers. Threads are partitioned into groups of w threads, called warps, and operate in lock-step. Warps are scheduled in a round robin manner, where each thread in the active warp is able to request a single memory access. Threads are restricted to a single active memory request, i.e., a thread cannot send another memory request until the previous request finishes. Data is stored across the w memory banks in a strided manner, i.e., data stored at memory address i is located in memory bank $i \pmod w$. The latency of a memory access is denoted l (i.e., time for a memory request to complete). Note that w memory banks implies a bandwidth of w, since each memory bank can only processes a single memory request at a time.

The difference between the DMM and UMM models is the way that the MMU is connected to the w memory banks. In the DMM, each memory bank has its own dedicated address line connecting to the MMU; and in the UMM, all memory banks share the same address line. Therefore, in the DMM, memory cells in different memory banks can be accessed in a single time step (analogous to bank conflict free access in shared memory). While in the UMM, groups of w consecutive memory cells, called address groups, can be accessed in a single time step (analogous to coalesced access in global memory).

1.2.2 Hierarchical Memory Machine (HMM)

The Hierarchical Memory Machine (HMM) model aims to model a whole GPU by combining a single UMM with multiple DMMs (Section 1.2.1) [83]. The HMM consists of d DMMs, each with d/p threads, and a single UMM of p threads (same threads used in the d DMMs). Each of the d DMMs execute independently, hence, are analogous to a streaming multiprocessor on a GPU. The latency of each of the DMMs are considered to be constant and the latency of the UMM is defined as l. The remaining parameters are the same as in Section 1.2.1.

1.2.3 Threaded Many-core Memory (TMM)

The Threaded Many-core Memory (TMM) model reflects the architecture of machines that are characterized by a large number of threads which are able to hide latency to memory by taking advantage of fast context switching of threads [75]. The authors argue that the number of memory accesses does not matter, as long as a sufficient number of threads are available to hide latency. Thus, the TMM model aims to bound the number of threads needed in order to sufficiently hide latency.

The TMM model consists of a 2-level memory hierarchy: a slow global memory and a fast local memory of size Z shared by a group of Q threads, called *core groups*. All threads in a core group execute in lock-step. Data elements in slow global memory are transferred to fast local memory in chunks of C contiguous elements with latency L. P is the total number of processors and X is the hardware maximum number of threads that can be scheduled onto a single core (due to various resource constraints such as number of registers).

The performance metrics in this model are: work, denoted T_1; span, denoted T_∞; total number of global memory chunk transfers (i.e., global memory transactions), denoted M; number of threads per core, denoted τ; and the amount of local memory used per thread, denoted S. Let T_E be the effective work (considers both computation and memory accesses), T_P be the runtime using P processors, and S_P be the speedup using P processors.

$$T_E = O\left(\texttt{max}\left(T_1, \frac{M \cdot L}{\tau}\right)\right)$$

$$T_P = O\left(\texttt{max}\left(\frac{T_E}{P}, T_\infty\right)\right) = O\left(\texttt{max}\left(\frac{T_1}{P}, T_\infty, \frac{M \cdot L}{\tau \cdot P}\right)\right)$$
$$S_P = \frac{T_1}{T_P} = \Omega\left(\texttt{min}\left(P, \frac{T_1}{T_\infty}, \frac{P \cdot T_1 \cdot \tau}{M \cdot L}\right)\right)$$

1.2.4 Abstract GPU (AGPU)

The Abstract GPU (AGPU) model consists of a host (CPU), which allows for the execution of device programs (i.e., kernels), and a device (GPU) [72]. The device (GPU) is composed of p processors, each capable of running a single thread. Processors are partitioned into groups of size b, called a *multiprocessor*, hence, there are a total of $k = p/b$ total multiprocessors. All processors in a multiprocessor executes in lock-step.

Each multiprocessor is connected to a large and slow global memory unit, which can be accessed by every multiprocessor and the host. The global memory unit is divided into blocks of b contiguous cells. Thus, when accessing a particular global memory location, all b elements of the particular block must be transferred. Additionally, each multiprocessor has its own small and fast *shared memory* unit of size M. Shared memory is divided into b banks, where all b processors can accesses a distinct bank in a single step. If multiple processors access the same bank, then the accesses are serialized, which is analogous to bank conflicts.

For an AGPU algorithm, the time complexity is the number of instructions executed per multiprocessor and the I/O complexity is the number of global memory accesses across all multiprocessors. Let m be the amount of shared memory used by a single multiprocessor. The efficiency of multithreading, called multiplicity, is defined as $\mathcal{M} = M/m$.

The AGPU model assumes that the number of threads and number of processors are equal. However, the authors provide a theorem to account for v virtual processors.

Theorem 1. *Let $v > p$.*

$$AGPU_{I/O}(p, b, M) = AGPU_{I/O}(v, b, M)$$
$$AGPU(p, b, M) \leq \left\lceil \frac{v}{p} \right\rceil AGPU(v, b, M)$$

where $AGPU_{I/O}(...)$ is the I/O complexity and $AGPU(...)$ is the time complexity.

1.2.5 Abstract Transferring GPU (ATGPU)

The Abstract Transferring GPU (ATGPU) model extends the AGPU model (Section 1.2.4) by considering the the cost of data transfer between the host (CPU) and device (GPU) [19]. Thus, the ATGPU model uses the same parameters as in the AGPU model in addition to G, which is the size of global memory.

An algorithm in the ATGPU model is divided into rounds. A round starts with data transferred from the host to the device, then the kernel is executed on the device, and finally the round ends with data transferred from the device to the host (and additional synchronization overhead). Let R be the total number of rounds of a particular algorithm, t_i be the maximum number of executed operations across all multiprocessors in round i, q_i be the total number of global memory accesses across all multiprocessors in round i, I_i be the number of words transferred from the host to device in round i, and O_i be the number of words transferred from the device to host in round i.

Different cost functions are defined in order to factor in the cost needed to perform various hardware instructions. Let γ be the cost for a multiprocessor to execute an instruction (e.g., clock rate), λ be the cost to access a global memory block, and σ be the cost to synchronize (at the end of each round). The cost function for host and device data transfer involves additional parameters. Let α be the initial overhead cost of data transfer transaction and β be the cost of sending a single word. Let $\hat{I}_i$ be the number of data transfer transactions from the host to device in round i and $\hat{O}_i$ be the number of data transfer transactions from the device to host in round i. Let $T_I(i) = \hat{I}_i\alpha + I_i\beta$ be the cost of data transfer from host to device in round i and $T_O(i) = \hat{O}_i\alpha + O_i\beta$ be the cost of data transfer from host to device in round i.

Therefore, the cost of the whole algorithm is defined as:

$$\sum_{i=1}^{R} T_I(i) + \frac{t_i + \lambda q_i}{\gamma} + T_O(i) + \sigma$$

Similar to the AGPU model, the ATGPU model also has a theorem to account for different number of processors.

Theorem 2. *Let $k' < k$*

$$\sum_{i=1}^{R} T_I(i) + \frac{\left\lceil \frac{k}{k'\ell} \right\rceil t_i + \lambda q_i}{\gamma} + T_O(i) + \sigma$$

where $\ell = \mathbf{min}\left(\left\lfloor \frac{M}{m} \right\rfloor, H\right)$ is the maximum number of concurrent blocks on a multiprocessor (H represents the hardware imposed limit).

1.3 Parallel Models of Computation

In this section, we provide a review of parallel models that we propose to use for developing and analyzing GPU algorithms. The Parallel Random Access Machine (PRAM) model (Section 1.3.1) models the overall parallelism and computational efficiency; the Parallel External Memory (PEM) model (Section 1.3.2) models the number of parallel coalesced accesses to global memory; and the Distributed Memory Machine (DMM) model (Section 1.3.3) models the number of parallel accesses to shared memory.

1.3.1 Parallel Random Access Machine (PRAM)

The Parallel Random Access Machine (PRAM) model [62] is a parallel extension of the well-known sequential RAM model. It consists of p processors connected to a shared memory space. Each processor is viewed as a sequential RAM machine and all processors execute in a synchronous manner (i.e., an implicit global synchronization is performed after every operation). All communication between processors are performed by reading from and writing into the shared memory space.

For an input of size n, there are two performance metrics in this model: (1) *work*, denoted $W(n)$, which is the total number of operations performed by all processors; and (2) *depth*, denoted $D(n)$, which is the maximum number of operations performed by any single processor if the algorithm is executed using infinite processors (also known as *span* or *critical path length*). The runtime using p processors can then be computed as $O\left(\frac{W(n)}{p} + D(n)\right)$, which is known as Brent's Scheduling Principle [13].

The PRAM model has 3 standard variants, based on the allowed shared memory mechanisms: EREW (Exclusive Read Exclusive Write), CREW (Concurrent Read Exclusive Write), and CRCW (Concurrent Read Concurrent Write). In the EREW PRAM, both concurrent read and write access is not allowed; in the CREW PRAM, concurrent read access is allowed and concurrent write access is not allowed; and in the CRCW PRAM, both concurrent read and write access is allowed. Additional CRCW PRAM models have been defined based on various concurrent write access mechanisms (i.e., how to resolve write conflicts). For example, in the common-CRCW PRAM, concurrent writes are only allowed if all processors are writing the same value to the same memory location; and in the arbitrary-CRCW PRAM, only a single arbitrary processor succeeds in writing its value and the remaining processors fail (i.e., do not write its value to the memory location). As concurrent writes are undefined on GPUs, we utilize the CREW PRAM model to analyze the available parallelism and computational complexity of GPU algorithms.

Historical Note: The PRAM model was formalized as early as 1978 by Fortune and Wyllie [45]. They denote the model as P-RAM and it is defined to be an unbounded set of processors connected to an unbounded shared memory space. One major difference in the models is that in the P-RAM model, processors can execute a FORK instruction, which allows the processor to start an inactive processor. A P-RAM algorithm is then defined to start initially on a single processor and parallelism is invoked via calling FORK. Therefore, in the P-RAM model, it takes $O(\log p)$ time to initialize p processors; while in the PRAM model, all p processors are active from the start of the algorithms.

1.3.2 Parallel External Memory (PEM)

The Parallel External Memory (PEM) model [4] is a parallel extension of the sequential External Memory model [2]. In the EM model, a processor contains fast internal memory of size M and

data initially resides in a large and slow external memory. To perform computation on data, that data must be transferred from external memory into the processors internal memory, using contiguous blocks of B data elements. The complexity metric of the EM model, *I/O complexity*, is the number of such blocks transferred during the algorithm. Similarly, in the PEM model, each of the P processors contains a private memory space of size M. However, external memory is now shared among all P processors. When performing computation on data, the data is still transferred between external memory and an individual processor's internal memory in blocks of size B. The performance metric in the PEM model is the parallel I/O complexity, which is the maximum number of blocks transferred by any one of the processors throughout the algorithm.

On a GPU, w contiguous global memory locations are able to be transferred to a group of w threads, called a *warp*. This behavior is equivalent to a single PEM processor accessing a block of w contiguous elements in global memory. In other words, by using $B = w$, each thread in a warp will accesses consecutive memory locations resulting in $O(1)$ global memory transactions. Thus, by mapping each PEM processor to a warp and setting $B = \Theta(w)$, we can utilize the PEM model to analyze the number of parallel coalesced global memory accesses on a GPU.

1.3.3 Distributed Memory Machine (DMM)

The Distributed Memory Machine (DMM) model (originally called the Module Parallel Machine model) considers data to be stored across a set of memory modules, rather than in a single shared memory space [76]. The motivation for this model was to reflect the architecture of the Ultracomputer, which featured p processors connected to p memory modules [53]; and to study the granularity of parallel memories problem. This problem studies the simulation of shared memory models on distributed models (e.g., the simulation of PRAM algorithms on the DMM).

The DMM model consists of p synchronous processors and p memory modules with a complete interconnection network between processors and memory modules. In each step of a DMM algorithm, processors are able to send a memory request to any of the p memory modules. However, each memory module is only able to respond to a single memory request at a time. Thus, multiple memory requests to a single memory module results in these memory requests being queued in an arbitrary order and processed sequentially. The cost of each step is the maximum number of requests across all memory modules.

In the DMM model, when and where each processor sends its memory request to is called the *access schedule*. An access schedule is called *simple* if it does not make "complicated" decisions and does not redistribute memory requests among the processors before sending them to memory modules. Furthermore, an access schedule is called *oblivious* if processors communicate independently of input keys and *non-oblivious* otherwise.

The DMM model has been mostly overlooked by the GPU community and has been reinvented with minor variations to model accesses in shared memory. Dotsenko *et al.* [36] visualized shared

memory as a 2-dimensional matrix; and Nakano [83] formalized this approach with the Discrete Memory Machine model (see Section 1.2.1). Afshani and Sitchinava [1] simplified the Discrete Memory Machine model by removing the latency parameter and considering a single warp. This simplification of the Discrete Memory Machine model is equivalent to the DMM model (with p being equal to the number of threads in a warp). These minor variations of the DMM have been used to analyze various algorithms such as: scanning [36], sorting [1, 67], searching [66], transposition [20], and permuting [1, 68].

Automatic Conflict Resolution: Historically, the DMM model has been used to study the *granularity of parallel memories* problem [26, 27, 28, 34, 63, 76, 81, 103]: given p arbitrary access to memory, the problem is to find the access schedule that results in the least amount of memory requests to any single memory module on a p processor DMM. In other words, this problem considers the simulation of an arbitrary PRAM algorithm step on the DMM. The current best solution is a non-oblivious access schedule that results in $O(\log \log \log p \log^* p)$ cost with high probability [27]. This approach relies on redundancy of data elements and universal hashing in order to reduce the expected number of requests to any single memory module.

1.4 Organization

In this dissertation, we use the PRAM model (Section 1.3.1), PEM model (Section 1.3.2), and DMM model (Section 1.3.3) to analyze and develop GPU algorithms. In Chapter 2, we study the problem of searching and use the PRAM and PEM models to analyze and develop parallel in-place permutations for constructing various implicit search tree layouts, which provide improved cache-efficiency when compared to binary search. We measure experimentally the cost of performing these permutations and a batch of queries, compared to binary search on a sorted array. Next in Chapter 3, we use the PRAM and PEM models and present a parallel priority queue, denoted parBucketHeap, and use it in a parallel variant of Dijkstra's algorithm for finding single source shortest paths (SSSP). We compare the performance of our SSSP implementation against the state-of-the-art GPU implementations for SSSP. In Chapter 4, we study the problem of pairwise merge on GPUs, which is the current fastest approach for comparison-based sorting on GPUs, and use the DMM model to prove that there exists inputs that cause the asymptotic worst-case number of bank conflicts in shared memory. Moreover, we show in practice that the bank conflicts incurred from our constructed worst-case inputs results in significant slowdown. Lastly, in Chapter 5, we present a bank conflict free algorithm for loading elements from shared memory into registers for balanced two-way divide-and-conquer algorithms. We experimentally show, via pairwise mergesort, that our bank conflict free approach eliminates the slowdown due to bank conflicts in practice.

CHAPTER 2
PARALLEL IN-PLACE CONSTRUCTION OF IMPLICIT SEARCH TREE LAYOUTS

Searching is a fundamental computational problem that arises in many applications. When many queries are expected to be performed, data is often stored in a data structure that is conducive to efficient searching. One such example are pointer-based search trees, e.g., a *binary search tree (BST)* is a binary tree such that for every vertex v, the key stored at v is greater than all the keys stored in the subtree rooted at v's left child and smaller than all the keys stored in the subtree rooted at v's right child. Pointer-based data structures, however, use at least a constant factor more space than the data itself, which can be prohibitive in limited-memory environments. In contrast, if the data is stored in sorted order, efficient search can be performed using binary search without using any extra space. The advantage of search trees lies in their efficient updates (insertions and deletions of elements). However, in the case of *static* data (i.e., data which will not change in the future), storing data in sorted order and performing binary search seems to be the preferred approach [69].

In this chapter, we study efficient *parallel* transformations of a static sorted array into various *implicit* search tree layouts (defined in Section 2.1.1) and the minimum number of queries needed to justify the extra time to perform such transformations in practice. Moreover, since binary search on already sorted data does not require any additional space, we require that these transformations be performed *in-place*.

2.1 Preliminaries

2.1.1 Memory Layouts of Static Search Trees

The *BST layout* is defined by the breadth-first left-to-right traversal of a complete binary search tree. Given the index i of a node v in the BST layout, the indices of the left and right children of v can be computed in $O(1)$ time as $2i + 1$ and $2i + 2$ (using 0-indexing), respectively. Figure 2.1 depicts an example 15-node BST layout.

A *complete B-tree* [8] is a complete multi-way search tree, where each node (except possibly the last leaf node) contains exactly B elements and every internal node (except possibly the last one) has exactly $B+1$ children. The *Level-order B-tree layout* is defined by the breadth-first left-to-right traversal of a complete B-tree. Figure 2.2 depicts the B-tree layout for $N = 26$ and $B = 2$.

The *van Emde Boas (vEB) layout* [92] is defined recursively as follows. The vEB layout of a tree with a single vertex is the vertex itself. Given a complete binary search tree $\mathcal{T}$ with N vertices and height $h = \lfloor \log N \rfloor > 0$, consider the top subtree $\mathcal{T}_0$ of height $\lfloor (h-1)/2 \rfloor$ containing $r = 2^{\lfloor (h-1)/2 \rfloor} - 1$ vertices, and $r+1$ bottom subtrees $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_{r+1}$, each of height $\lceil (h-1)/2 \rceil$ and

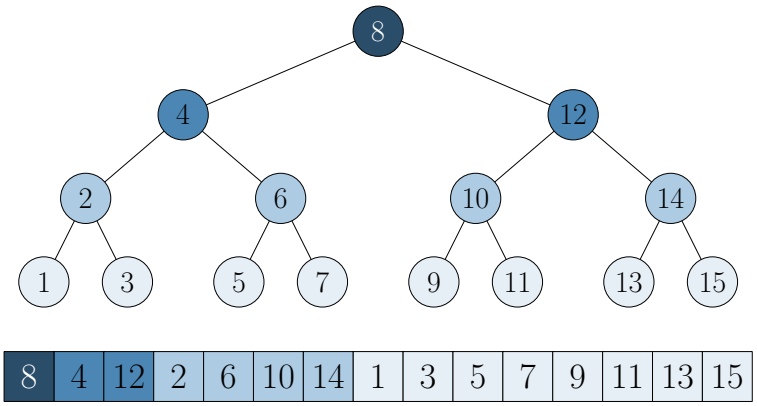

Figure 2.1: BST layout for $N = 15$.

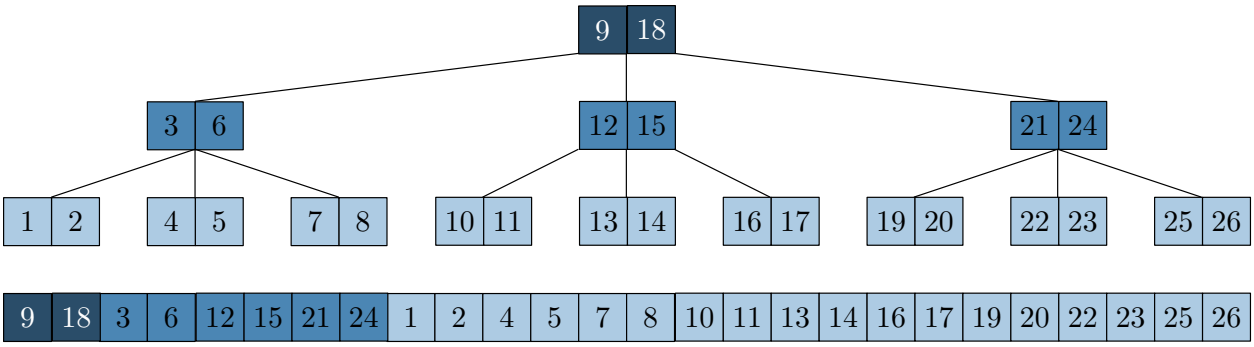

Figure 2.2: Level-order B-tree layout for $N = 26$ and $B = 2$.

each rooted at the children of the $(r+1)/2$ leaves of $\mathcal{T}_0$. The vEB layout of $\mathcal{T}$ is defined recursively as the vEB layout of $\mathcal{T}_0$, followed by the vEB layouts of each $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_{r+1}$. Figure 2.3 depicts an example vEB layout with 15 nodes.

The above definition of the vEB layout for $N \neq 2^{h+1}-1$ complicates the permutation algorithms described in Sections 2.3.3 and 2.4.1 because the number of vertices in each bottom subtree may be different. Instead, in this work, we modify the definition of the vEB layout for $N \neq 2^{h+1} - 1$ as follows. Let $r = 2^{\lfloor (h-1)/2 \rfloor} - 1$ and $l = 2^{\lceil (h-1)/2 \rceil} - 1$. The top subtree $\mathcal{T}_0$ of the vEB layout will always contain r vertices and the remaining $N - r$ elements will form $x = \lceil (N-r)/l \rceil$ bottom subtrees, $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_x$. Each of the first $y = \lfloor (N-r)/l \rfloor$ bottom subtrees, $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_y$, will consist of exactly l vertices. If $N - r$ is not a multiple of l, i.e., $x = y + 1$, then the last bottom subtree, $\mathcal{T}_x$, will contain $1 \leq l' < l$ vertices. As in the standard definition, the vEB layout consists of $\mathcal{T}_0$, immediately followed by $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_x$, with each subtree laid out recursively ($\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_y$ uses the standard vEB layout and if $x = y + 1$, then $\mathcal{T}_x$ uses this modified approach).

In our definition of the vEB layout, at each recursive level, there is at most one bottom subtree that contains a different number of vertices than all other bottom subtrees and if it exists, is

always located at the end of the layout. This observation allows us to easily adapt the permutation algorithms described in Sections 2.3.3 and 2.4.1 and query optimizations described in Section 2.7.1 to work with arrays of sizes $N \neq 2^{h+1} - 1$, without affecting the asymptotic analysis.

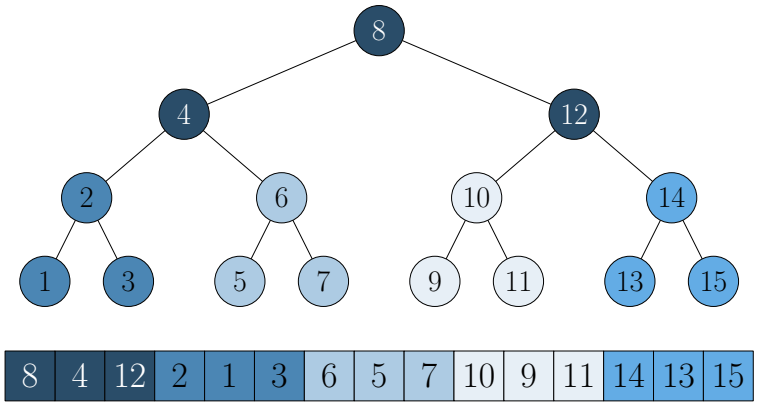

Figure 2.3: van Emde Boas (vEB) layout for $N = 15$.

The I/O complexity of performing a search query on an array of size N in the BST layout is $O(\log(N/B))$, and $\Theta(\log_B N)$ in the B-tree and vEB layouts [14, 92]. In theory, because the definition of the vEB layout does not make use of the parameter B, i.e., it is *cache-oblivious* [49], querying the vEB layout on architectures with multiple levels of cache will result in the asymptotically optimal number of accesses at every level of the memory hierarchy [92].

2.1.2 Previous Work on Permutations

The transformation from sorted order to an implicit search tree layout is a special case of permuting an array of N elements. Let $\pi : [N] \to [N]$ be an arbitrary permutation. For the purpose of this paper, we assume that π is given as a function that can be described concisely in $O(1)$ space (e.g., not as a table that explicitly gives $\pi(i)$ for each i). Let τ_π be the time it takes to evaluate $\pi(i)$. For example, while $\tau_\pi = O(1)$ for the BST and B-tree layouts, it is not obvious how to compute $\pi(i)$ faster than $O(\log \log N)$ time for the vEB layout.

Note that for the problem of permuting N elements using P processors, $\Omega((N/P) \cdot \tau_\pi)$ is the trivial lower bound in the PRAM model. If there is no in-place requirement, any permutation π can be implemented in $O(\lceil N/P \rceil \cdot \tau_\pi)$ time in parallel: each entry $A[i]$ can be copied to $B[\pi(i)]$ independently of each other. Thus, the BST and B-tree layouts can be constructed from sorted data in $O(\lceil N/P \rceil)$ time and the vEB layout can be constructed in $O(\lceil N/P \rceil \log \log N)$ time.

It is well known that every permutation can be decomposed into disjoint cycles. A cyclic permutation can be implemented sequentially in-place trivially by starting at a single vertex and following the cycle. However, for a general permutation this approach still needs additional space to mark the elements that have already been permuted, unless it can identify all disjoint cycles up

front.

When it comes to *in-place* permutations, Fich *et al.* [43] showed that every permutation π can be implemented sequentially in-place in $O((N \log N) \cdot \tau_\pi)$ time. For a special case when the data is permuted from a sorted order, they observed that they can check if an element has already been moved by computing the inverse permutation π^{-1} to determine if the element is not in its original sorted order. Thus, for this special case, the time can be reduced to $O(N \cdot (\tau_\pi + \tau_{\pi^{-1}}))$. However, it is not obvious how to parallelize their algorithm, nor is it trivial to compute π^{-1} for the vEB layout.

Yang *et al.* [111] observed that every permutation is the product of two involutions. A permutation π is an *involution* if it is its own inverse, i.e., $\pi(\pi(i)) = i$ for all i. Moreover, every involution is composed of disjoint cycles of length at most 2, i.e., can be implemented in parallel and in-place by swapping pairs of elements. Thus, if the two involutions of a permutation are known, this permutation can be implemented in parallel and in-place. This result is non-constructive, i.e., given an arbitrary permutation π it is not clear how to determine the two involutions that define π; however, the authors show how to determine the involutions of a cyclic permutation.

One permutation of particular interest for this work is the *perfect shuffle* [33]: a permutation in which two lists of equal length are interleaved perfectly. A generalization is the *k-way perfect shuffle*, where k equal-length lists are interleaved perfectly [93]. These permutations have many applications (e.g., parallel processing [99], Fast Fourier Transforms (FFT) [30, 99], Kronecker products [30, 32], encryption [101], sorting [99], and merging [29, 41, 42]). Ellis *et al.* [39, 40] use a number-theoretic approach to compute representative elements of the disjoint cycles of the perfect shuffle and the k-way perfect shuffle, thus making a sequential in-place approach possible. Jain [61] relies on the fact that 2 is primitive root of 3^k for any $k \geq 1$, which makes it possible to compute the representative elements of the disjoint cycles recursively for any N. Finally, Yang *et al.* [111] use the product of involutions approach and describe the involutions for the k-way perfect shuffle for two cases: (i) $N = k^d$ and (ii) $N = kd$ for some integer $d > 1$. For (i), the involutions involve reversing the base-k representation of element indices. For (ii), the involutions involve computing modular inverses of element indices and finding greatest common divisors. We use these results of Yang *et al.* [111] for designing our involution-based permutation algorithms.

2.1.3 Parallel In-place Computations

There is a bit of ambiguity in the literature when it comes to the definition of *in-place* algorithms. Strictly speaking, a (sequential) algorithm is said to be *in-place* if it uses at most $\Theta(1)$ additional space (a processor needs at least one register to perform any useful work) [42]. However, for a recursive algorithm, at least $\Omega(\log N)$ additional space is needed to implement the recursion stack of a balanced recursion. Therefore, it is reasonable for an in-place algorithm to use up to $O(\log N)$ additional space, although often such algorithms are called *in-situ* [41, 71]. When it comes to

parallel algorithms, there is an additional complication. Each of the P processors needs to have $\Omega(1)$ space to perform any meaningful work. Moreover, for asynchronous recursion, $\Omega(\log N)$ space is needed per processor, i.e., a total of $\Omega(P \log N)$ additional space. Therefore, if $P = \frac{N}{\log N}$, the total additional space becomes $\Omega(N)$ and trivially non-in-place algorithms could be viewed as being in-place. To avoid this situation, we define *in-place parallel* computation as follows:

Definition 3. *A parallel algorithm running on P processors each having an internal memory of size M is called* in-place *if it uses at most $O(P(M + \log N))$ additional space and works correctly for any $P \geq 1$ processors.*

In the PRAM model, $M = O(1)$ is the number of registers per processor so it reduces to $O(P \log N)$; while in the PEM model, M is the size of each processor's internal memory. Note that the requirement for an algorithm to work correctly for any $P \geq 1$ precludes the view of trivially non-in-place algorithms designed for large P as being in-place.

2.2 Contributions

We present parallel algorithms for the in-place permutation of a sorted array into the BST, B-tree, and vEB layouts, and analyze their time and I/O complexities. We propose two types of algorithms:

1. Building on the work of Yang *et al.* [111] and Fich *et al.* [43], we determine the pairs of *involutions* required to permute a sorted array into the BST layout. We also determine the $\log_{B+1} N$ pairs of involutions required to permute a sorted array into the B-tree layout. The B-tree involutions can be used in order to permute a sorted array into the vEB layout.

2. Using a *cycle-leader* approach, we develop an efficient parallel in-place algorithm to permute a sorted array into the vEB layout. By recursively applying this approach, we are able to design algorithms for permuting a sorted array into the B-tree layout. The B-tree layout algorithm can be used to obtain the BST layout by setting $B = 1$.

The involution-based approach entails reversing a subset of the digits of numbers represented in an arbitrary base-k (for BST $k = 2$, for B-tree $k = B + 1$). If implemented in software, the worst-case complexity of this operation is linear with the number of digits in the base-k representation of the integer N being reversed, i.e., $O(\log_k N)$. Some architectures provide it as a built-in hardware primitive (i.e., it takes $O(1)$ time), in particular, NVIDIA GPUs implement this operation in hardware for $k = 2$. We parameterize the time of this operation as $T_{REV_k}(N)$.

To the best of our knowledge, our algorithms are the first parallel in-place algorithms for permuting a sorted array into the considered search tree layouts. The time and I/O complexities of our algorithms are summarized in Table 2.1. Our cycle-leader algorithms exhibit better I/O complexity, while our involution-based algorithms are much simpler and trivial to parallelize. We

Table 2.1: Asymptotic time and I/O complexity bounds of each of our algorithms. N is the input size, P is the number of processors, M and B are the sizes of the internal memory and the transfer block, respectively, in the PEM model. $K = \text{MIN}(\frac{N}{P}, M)$ and $T_{REV_k}(N)$ is the time complexity of reversing the digits of number N in the base-k representation.

Algorithm	Time complexity	I/O complexity
Involution BST	$O\left(\frac{N}{P} \cdot T_{REV_2}(N)\right)$	$O\left(\frac{N}{P}\right)$
Involution B-tree	$O\left(\left(\frac{N}{P} + \log_{B+1} N\right) \log N\right)$	$O\left(\frac{N}{P} + B \log_{B+1} \frac{N}{K}\right)$
Involution vEB	$O\left(\frac{N}{P} \log N\right)$	$O\left(\frac{N}{P} \log \log_K N\right)$
Cycle-leader BST	$O\left(\left(\frac{N}{P} + \log N\right) \log N\right)$	$O\left(\left(\frac{N}{PB} + \log \frac{N}{K}\right) \log \frac{N}{K}\right)$
Cycle-leader B-tree	$O\left(\left(\frac{N}{P} + \log_{B+1} N\right) \log_{B+1} N\right)$	$O\left(\left(\frac{N}{PB} + \log_{B+1} \frac{N}{K}\right) \log_{B+1} \frac{N}{K}\right)$
Cycle-leader vEB	$O\left(\frac{N}{P} \log \log N\right)$	$O\left(\frac{N}{PB} \log \log_K N\right)$

evaluate these algorithms experimentally and find that, compared to a binary search on non-permuted input, the permutation overhead of our permutation algorithms is offset by the query time for as few as $0.013N$ on a NVIDIA GPU.

The remainder of this chapter is organized as follows. Section 2.3 presents our involution-based algorithms and Section 2.4 presents our cycle-leader algorithms, each section analyzing the time complexity of these algorithms. For ease of exposition, in Sections 2.3 and 2.4 we consider only *perfect* trees, i.e., complete trees in which every level is full. Section 2.5 analyzes the I/O complexity of our algorithms. Section 2.6 discusses extensions of our algorithms to non-perfect trees. Section 2.7 goes over experimental optimizations and Section 2.8 presents experimental results. Finally, Section 2.9 concludes with a summary.

2.3 Involution Approach

2.3.1 BST Layout

A perfect BST contains $N = 2^d - 1$ vertices. Fich *et al.* [43] propose a sequential in-place algorithm to permute a sorted array into the BST layout. They note that the permutation satisfies the property that for a given index $i = (x10^j)_2$ in binary representation, the index of that element in the BST layout is $\pi(i) = (0^j 1x)_2$. Let $\text{REV}_k(b, i)$ be the operation that reverses the b least significant digits of the base-k representation of the integer i. The previously mentioned permutation can be computed as $\pi(i) = \text{REV}_2(d-(j+1), (\text{REV}_2(d, i))$. Since REV_2 is an involution [43], we can perform the permutation π in parallel in just two rounds of $O(N)$ independent swaps.

The time to compute $\pi(i)$ depends on the time to perform the REV_2 operation. Thus, this algorithm has depth $D(N) = O(T_{\text{REV}_2}(N))$ and work $W(N) = O(N \cdot T_{\text{REV}_2}(N))$.

2.3.2 B-tree Layout

The B-tree layout algorithm relies on the k-way perfect shuffle involution approach developed by Yang *et al.* [111]. Let us first review their results.

Let $J_r(i) = g \cdot (r \cdot (\frac{i}{g})^{-1} \pmod{\frac{N-1}{g}})$ where g is the greatest common divisor of i and $N-1$. Yang *et al.* [111] show that for $N = k^d$ and $N = kd$ the k-way perfect shuffle can be implemented as $\Xi_1(i) = \textsc{rev}_k(d, \textsc{rev}_k(d-1, i))$ and $\Xi_2(i) = J_k(J_1(i))$, respectively, for any integer $d > 1$. We note that the k-way "un-shuffle", which we use, can be performed by simply reversing the order in which the involutions are performed.

A perfect B-tree has $N = (B+1)^d - 1$ elements, for some $d > 1$. Since each leaf node contains B contiguous elements from the sorted array, every $(B+1)$-th element is stored in a non-leaf (i.e., internal) node. Let S_i, for $i \in \{0, 1, 2, ..., B\}$, denote the list of elements at locations $i + j(B+1)$, for $j \in \{0, 1, ..., \left\lfloor \frac{N}{(B+1)} \right\rfloor\}$. In other words, each S_i is comprised of the elements starting at i, strided by $B+1$. By this definition, S_B contains all internal elements and S_l, for $0 \leq l \leq B-1$, contains the l-th element of each leaf node. We first perform the $(B+1)$-way perfect un-shuffle (via Ξ_1 while using or simulating 1-indexing), which will gather each S_i into contiguous space and lay them out in sequence. We then apply the B-way perfect shuffle (via Ξ_2 while using or simulating 0-indexing) on all S_l lists to interleave the leaf elements back into their corresponding leaf nodes, i.e., into their correct positions. All leaf elements are thus correctly permuted and we recurse on S_B.

Recall that $\textsc{rev}_k$ can take up to $O(\log_k N)$ time. Finding the modular inverse, however, requires using the extended Euclidean algorithm [111], which takes $O(\log N)$ time. The latter dominates the running time, resulting in $O(\log N)$ time for both operations. The work and depth complexities of our B-tree permutation algorithm are given in Proposition 4 and 5, respectively.

Proposition 4.

$$\begin{aligned} W(N) &= W\left(\frac{N}{B+1}\right) + O(N \log N) \\ &= O(N \log N)\ . \end{aligned}$$

Proof. Guess: $W(N) = c\left(\frac{B+1}{B}\right) N \log N - c\left(\frac{B+1}{B^2}\right) N \log (B+1)$.

$$\begin{aligned} W\left(\frac{N}{B+1}\right) &= c\left(\frac{B+1}{B}\right)\left(\frac{N}{B+1}\right) \log \frac{N}{B+1} - c\left(\frac{B+1}{B^2}\right)\left(\frac{N}{B+1}\right) \log (B+1) \\ &= c\left(\frac{1}{B}\right) N \log N - c\left(\frac{1}{B}\right) N \log (B+1) - c\left(\frac{1}{B^2}\right) N \log (B+1) \\ &= c\left(\frac{1}{B}\right) N \log N - c\left(\frac{1}{B} + \frac{1}{B^2}\right) N \log (B+1) \\ &= c\left(\frac{1}{B}\right) N \log N - c\left(\frac{B}{B^2} + \frac{1}{B^2}\right) N \log (B+1) \end{aligned}$$

$$
\begin{aligned}
&= c\left(\frac{1}{B}\right) N \log N - c\left(\frac{B+1}{B^2}\right) N \log\left(B+1\right) , \\
W(N) &= W\left(\frac{N}{B+1}\right) + cN \log N \\
&= c\left(\frac{1}{B}\right) N \log N - c\left(\frac{B+1}{B^2}\right) N \log\left(B+1\right) + cN \log N \\
&= c\left(\frac{1}{B}+1\right) N \log N - c\left(\frac{B+1}{B^2}\right) N \log\left(B+1\right) \\
&= c\left(\frac{1}{B}+\frac{B}{B}\right) N \log N - c\left(\frac{B+1}{B^2}\right) N \log\left(B+1\right) \\
&= c\left(\frac{B+1}{B}\right) N \log N - c\left(\frac{B+1}{B^2}\right) N \log\left(B+1\right) .
\end{aligned}
$$

Therefore, $W(N) = O(N \log N)$. □

Proposition 5.

$$
\begin{aligned}
D(N) &= D\left(\frac{N}{B+1}\right) + O(\log N) \\
&= O(\log_{B+1} N \cdot \log N) .
\end{aligned}
$$

Proof. Let $h = \log_{B+1} N$.
Guess: $D(N) = c(h+1)\log N - \frac{c}{2}h^2 \log\left(B+1\right) - \frac{c}{2}h \log\left(B+1\right)$.

$$
\begin{aligned}
D(N) &= D\left(\frac{N}{B+1}\right) + c \log N \\
&= ch(\log N - \log\left(B+1\right)) - \frac{c}{2}(h-1)^2 \log\left(B+1\right) - \frac{c}{2}(h-1)\log\left(B+1\right) + c \log N \\
&= c(h+1)\log N - ch \log\left(B+1\right) - \frac{c}{2}(h^2 - 2h + 1)\log\left(B+1\right) - \frac{c}{2}h \log\left(B+1\right) + \frac{c}{2}\log\left(B+1\right) \\
&= c(h+1)\log N - \frac{c}{2}h^2 \log\left(B+1\right) - \frac{c}{2}h \log\left(B+1\right) .
\end{aligned}
$$

Therefore, $D(N) = O(\log_{B+1} N \log N)$. □

2.3.3 van Emde Boas Layout

We are able to apply the B-tree layout algorithm for the vEB layout of height h, by using $B = 2^{\lceil (h-1)/2 \rceil} - 1$ and recursing on each subtree of the vEB layout. The resulting work and depth complexities are:

Proposition 6.

$$
W(N) = \sqrt{N} \cdot W\left(\sqrt{N}\right) + O(N \log N)
$$

$$= O(N \log N) \ .$$

Proof. Guess: $W(N) = 2cN \log N$

$$\begin{aligned} W(N) &= \sqrt{N}(2c\sqrt{N} \log \sqrt{N}) + cN \log N \\ &= cN \log N + cN \log N \\ &= 2cN \log N \ . \end{aligned}$$

Therefore, $W(N) = O(N \log N)$. □

Proposition 7.

$$\begin{aligned} D(N) &= D\left(\sqrt{N}\right) + O(\log N) \\ &= O(\log N) \ . \end{aligned}$$

Proof. Guess: $D(N) = 2c \log N$

$$\begin{aligned} D(N) &= 2c \log \sqrt{N} + c \log N \\ &= c \log N + c \log N \\ &= 2c \log N \ . \end{aligned}$$

Therefore, $D(N) = O(\log N)$. □

2.4 Cycle-leader Approach

2.4.1 van Emde Boas Layout

Recall from Section 2.1.1 that we define $\mathcal{T}_i$ as the i-th subtree of size $O(\sqrt{N})$: $\mathcal{T}_0$ is the "root" subtree consisting of $r = 2^{\lfloor (h-1)/2 \rfloor} - 1$ vertices of the upper $\left\lfloor \frac{h-1}{2} \right\rfloor$ levels, where $h = \lfloor \log N \rfloor$, while $\mathcal{T}_1, \ldots, \mathcal{T}_{r+1}$ are "leaf" subtrees consisting of $l = 2^{\lceil (h-1)/2 \rceil} - 1$ vertices each. Let $A[a_i : b_i]$ be the interval within the input array where the elements of $\mathcal{T}_i$ should be moved to. In particular, $a_0 = 1$, $b_0 = r$ and for all $1 \leq j \leq r+1$, $a_j = r + (j-1)l + 1$ and $b_j = r + jl$. Our algorithm first moves each $\mathcal{T}_i$ into $A[a_i : b_i]$, which we call the *equidistant gather* operation, then recursively permutes each $A[a_i : b_i]$ into the vEB layout. (Our equidistant gather operation is general enough to work for any $r \leq l$.)

We use $\mathcal{T}_i[a : b]$ to denote the subset of nodes of $\mathcal{T}_i$ from the a-th smallest to the b-th smallest in the sorted order. E.g., $\mathcal{T}_i[1 : k]$ represents the first k smallest elements of $\mathcal{T}_i$.

The following proposition bounds the range in the input array, where the elements of the leaf subtrees $\mathcal{T}_j$, for $j \geq 1$, are initially located:

Proposition 8. *For all $i = r - j + 2$, $1 \leq j \leq r + 1$, $\mathcal{T}_j[i : l]$ are already in their destination interval $A[a_j : b_j]$. If $i > l$, then no elements of $\mathcal{T}_j$ are in their destination interval.*

Proof. Since the input is in sorted order, for all $1 \leq i, j \leq l$, $\mathcal{T}_j[i]$ is initially located at index $i_{orig} = (j-1)(l+1) + i$. Hence, we check if $i_{orig} \geq a_j = r + (j-1)l + 1$. Solving for i results in $i \geq r - j + 2$. □

From the above proposition, we know that $\mathcal{T}_1[r+1 : l], \mathcal{T}_2[r : l], ..., \mathcal{T}_{r+1}[1 : l]$ are already in their destination intervals and only $\mathcal{T}_1[1 : r], ..., \mathcal{T}_r[1]$ need to be moved.

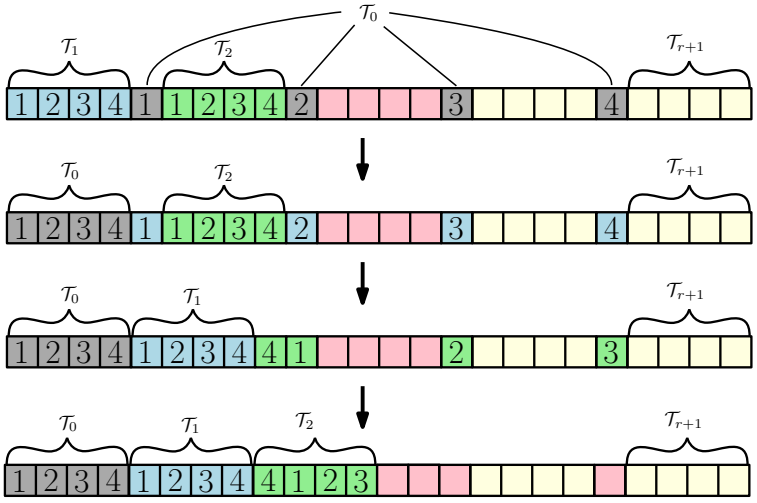

Figure 2.4: Illustration of the series of swaps needed to sequentially perform the equidistant gather operation for $r = l$.

We first consider a sequential strategy to perform the equidistant gather in-place: we perform r rounds of swapping, where after round i, all elements in the subtree $\mathcal{T}_i$ are in $A[a_i : b_i]$. Figure 2.4 illustrates the first few rounds of swapping for $r = l$. We see that, initially, all elements of $\mathcal{T}_0$ are distributed throughout the array. After the first round, $\mathcal{T}_0$ is in $A[a_0 : b_0]$ and $\mathcal{T}_1$ becomes distributed throughout the array. After repeating this process r times, each $\mathcal{T}_i$ is in $A[a_i : b_i]$, however, the elements in each $\mathcal{T}_i$ may not be in sorted order. Specifically, we need to perform a circular shift to the right by $r + 1 - i$ places (or equivalently $l - (r + 1 - i)$ to the left) on each $\mathcal{T}_i$.

We can parallelize this algorithm by unrolling the r sequential swap rounds and identifying the resulting disjoint cycles. We identify r disjoint cycles of the following form:

$$\begin{aligned}
&\mathcal{T}_0[1] \mapsto \mathcal{T}_1[1] \,, \\
&\mathcal{T}_0[2] \mapsto \mathcal{T}_1[2] \mapsto \mathcal{T}_2[1] \,, \\
&\mathcal{T}_0[3] \mapsto \mathcal{T}_1[3] \mapsto \mathcal{T}_2[2] \mapsto \mathcal{T}_3[1] \,, \\
&\quad \vdots
\end{aligned}$$

$$\mathcal{T}_0[r] \mapsto \mathcal{T}_1[r] \mapsto ... \mapsto \mathcal{T}_{r-1}[2] \mapsto \mathcal{T}_r[1] \ .$$

Since we can identify each element in each disjoint cycle, its position in the cycle, and the length of the cycle, as mentioned in Section 2.1.2, we can implement circular shifts in parallel and in-place in $O(1)$ depth and $O(N)$ work using the involutions of Yang *et al.* [111]. Therefore, since both stages of our algorithm are comprised of disjoint circular shifts, we can perform the equidistant gather in $O(1)$ time and $O(N)$ work. The work and depth complexities of this algorithm are:

Proposition 9.

$$\begin{aligned} W(N) &= \sqrt{N} \cdot W(\sqrt{N}) + O(N) \\ &= O(N \log\log N) \ . \end{aligned}$$

Proof. Guess: $W(N) = cN \log\log N$.

$$\begin{aligned} W(\sqrt{N}) &= c\sqrt{N} \log\log \sqrt{N} \\ &= c\sqrt{N} \log\left(\frac{1}{2} \log N\right) \\ &= c\sqrt{N} \log(2^{-1}) + c\sqrt{N} \log\log N \\ &= -c\sqrt{N} + c\sqrt{N} \log\log N \ , \\ W(N) &= \sqrt{N} \cdot W(\sqrt{N}) + cN \\ &= \sqrt{N}(-c\sqrt{N} + c\sqrt{N} \log\log N) + cN \\ &= -cN + cN \log\log N + cN \\ &= cN \log\log N \ . \end{aligned}$$

Therefore, $W(N) = O(N \log\log N)$ □

Proposition 10.

$$\begin{aligned} D(N) &= D(\sqrt{N}) + O(1) \\ &= O(\log\log N) \ . \end{aligned}$$

Proof. Guess: $D(N) = c \log\log N$.

$$\begin{aligned} D(\sqrt{N}) &= c \log\log \sqrt{N} \\ &= c \log\left(\frac{1}{2} \log N\right) \\ &= c \log 2^{-1} + c \log\log N \\ &= -c + c \log\log N \ , \end{aligned}$$

$$\begin{aligned} D(N) &= D(\sqrt{N}) + c \\ &= -c + c \log\log N + c \\ &= c \log\log N \ . \end{aligned}$$

Therefore, $D(N) = O(\log\log N)$. □

2.4.2 B-tree Layout

The idea is similar to the above vEB cycle-leader approach, except we have $r = \left\lfloor \frac{N}{(B+1)} \right\rfloor$ and $l = B$. Therefore, we need to extend the equidistant gather operation for $r > l$. We call this version the *extended equidistant gather* operation.

In a perfect B-tree of height h, $N = (B+1)^{h+1} - 1$. Let $C = \left\lceil \frac{N}{(B+1)^2} \right\rceil$. To perform the extended equidistant gather, we partition the array into $(B+1)$ partitions, where each partition will contain C internal elements (except for the first one, which will contain $C-1$ internal elements) and BC leaf elements. We move the internal elements of each partition to the front of that partition by applying the extended equidistant gather recursively on each partition. We then move the internal elements to the front of the whole array by applying the equidistant gather while treating each chunk of C elements as a single unit, and while ignoring the first $C-1$ internal elements of the first partition. At the base case of the recursion $C = 1$ and we can apply the equidistant gather directly to bring the internal elements to the front.

Since the equidistant gather takes $O(N)$ work and $O(1)$ depth, the extended equidistant gather takes:

Proposition 11.

$$\begin{aligned} W'(N) &= (B+1) \cdot W'\left(\frac{N}{B+1}\right) + O(N) \\ &= O(N \log_{B+1} N) \end{aligned}$$

Proof. Let $h = \log_{B+1} N$.
Guess: $W'(N) = cNh$.

$$\begin{aligned} W'\left(\frac{N}{B+1}\right) &= \frac{cN}{B+1}(h-1) \\ &= \frac{cNh}{B+1} - \frac{cN}{B+1} \ , \\ W'(N) &= (B+1) \cdot W'\left(\frac{N}{B+1}\right) + cN \\ &= (B+1)\left(\frac{cNh}{B+1} - \frac{cN}{B+1}\right) + cN \end{aligned}$$

$$= cNh - cN + ch$$
$$= cNh\ .$$

Therefore, $W'(N) = O(Nh) = O(N \log_{B+1} N)$. □

Proposition 12.

$$D'(N) = D'\left(\frac{N}{B+1}\right) + O(1)$$
$$= O(\log_{B+1} N)$$

Proof. Let $h = \log_{B+1} N$.
Guess: $D'(N) = ch$.

$$D'(N) = D'\left(\frac{N}{B+1}\right) + c$$
$$= c(h-1) + c$$
$$= ch\ .$$

Therefore, $D'(N) = O(h) = O(\log_{B+1} N)$. □

Once all the internal elements are gathered to the front of the array, we recursive on the internal elements, resulting in the following complexities:

Proposition 13.

$$W(N) = W\left(\frac{N}{B+1}\right) + W'(N)$$
$$= W\left(\frac{N}{B+1}\right) + O(N \log_{B+1} N)$$
$$= O(N \log_{B+1} N)$$

Proof. Guess: $W(N) = \frac{c(B+1)}{B} N \log_{B+1} N - \frac{c(B+1)}{B^2} N$.

$$\begin{aligned} W\left(\frac{N}{B+1}\right) &= \frac{c(B+1)}{B} \cdot \frac{N}{B+1} \cdot \log_{B+1} \frac{N}{B+1} - \frac{c(B+1)}{B^2} \cdot \frac{N}{B+1} \\ &= \frac{c}{B} N(\log_{B+1} N - 1) - \frac{c}{B^2} N \\ &= \frac{c}{B} N \log_{B+1} N - \frac{c}{B} N - \frac{c}{B^2} N \\ &= \frac{c}{B} N \log_{B+1} N - Nc\left(\frac{1}{B} + \frac{1}{B^2}\right) \end{aligned}$$

$$
\begin{aligned}
&= \frac{c}{B} N \log_{B+1} N - Nc\frac{B+1}{B^2} \,, \\
W(N) &= W\left(\frac{N}{B+1}\right) + cN \log_{B+1} N \\
&= \frac{c}{B} N \log_{B+1} N - N\frac{c(B+1)}{B^2} + cN \log_{B+1} N \\
&= \frac{c(B+1)}{B} N \log_{B+1} N - N\frac{c(B+1)}{B^2} \,.
\end{aligned}
$$

Therefore, $W(N) = O(N \log_{B+1} N)$.

□

Proposition 14.

$$
\begin{aligned}
D(N) &= D\left(\frac{N}{B+1}\right) + D'(N) \\
&= D\left(\frac{N}{B+1}\right) + O(\log_{B+1} N) \\
&= O(\log^2_{B+1} N)
\end{aligned}
$$

Proof. Let $h = \log_{B+1} N$.
Guess: $D(N) = \frac{c}{2}h^2 + \frac{c}{2}h$.

$$
\begin{aligned}
D\left(\frac{N}{B+1}\right) &= \frac{c}{2}(h-1)^2 + \frac{c}{2}(h-1) \\
&= \frac{c}{2}(h^2 - 2h + 1) + \frac{c}{2}h - \frac{c}{2} \\
&= \frac{c}{2}h^2 - ch + \frac{c}{2} + \frac{c}{2}h - \frac{c}{2} \\
&= \frac{c}{2}h^2 - \frac{c}{2}h \,, \\
D(N) &= D\left(\frac{N}{B+1}\right) + ch \\
&= \frac{c}{2}h^2 - \frac{c}{2}h + ch \\
&= \frac{c}{2}h^2 + \frac{c}{2}h \,.
\end{aligned}
$$

Therefore, $D(N) = O(h^2) = O(\log^2_{B+1} N)$. □

2.4.3 BST Layout

We can apply the B-tree cycle leader algorithm (Section 2.4.2) to the BST layout by setting $B = 1$, resulting in $O(N \log N)$ work and $O(\log^2 N)$ depth. Although this is worse than the involution-based algorithm from Section 2.3.1, the cycle-leader algorithm exhibits better spatial locality, which

we analyze in the next section.

2.5 I/O Optimizations

In this section, we analyze the I/O complexity of our proposed algorithms in the *parallel external memory (PEM)* model [4] – a parallel extension of the EM model. When applicable, we present additional modifications to the algorithms to improve the I/O efficiency.

Let $K = \text{MIN}\left(\frac{N}{P}, M\right)$ and assume that $P \leq \frac{N}{B}$, i.e., each processor processes at least one block, and $M \geq 2B + O(1)$, i.e., each processor can swap at least two blocks. In Section 2.5.2, we increase this assumption to $M \geq B^2$ (standard tall-cache assumption) and consequently $P \leq \frac{N}{B^2}$.

2.5.1 Involution-based Algorithms

We first consider the involution-based algorithms described in Section 2.3. The swaps performed by these algorithms can be an arbitrary distance away from each other. Hence, in the worst case these algorithms will perform $O(1)$ I/Os per swap. Thus, each iteration of an involution performs $O(\frac{N}{P})$ I/Os. For the B-tree and vEB layouts, however, once the subproblem is of size M or less, it will fit in internal memory. Proposition 15 and 16 provides the I/O complexity of the B-tree layout and vEB layout, respectively.

Proposition 15.

$$\begin{aligned} Q(N,P) &= \begin{cases} O\left(N/B\right) & \text{if } N \leq M \text{ and } P = 1 \\ Q\left(\frac{N}{B+1}, \text{MIN}\left(P, \frac{N}{B(B+1)}\right)\right) + O\left(\frac{N}{P}\right) & \text{otherwise} \end{cases} \\ &= O\left(\frac{N}{P} + B\log_{B+1}\frac{N}{K}\right) , \end{aligned}$$

Proof. Case 1: Suppose $P < \frac{N}{B(B+1)}$.
Guess: $Q(N,P) = c\left(\frac{B+1}{B}\right)\frac{N}{P}$.

$$\begin{aligned} Q(N,P) &= Q\left(\frac{N}{B+1}, P\right) + \frac{cN}{P} \\ &= c\left(\frac{B+1}{B}\right)\left(\frac{N}{P(B+1)}\right) + \frac{cN}{P} \\ &= \frac{cN}{PB} + \frac{cN}{P} \\ &= \frac{cN}{P}\left(\frac{1}{B} + 1\right) \\ &= \frac{cN}{P}\left(\frac{B+1}{B}\right) . \end{aligned}$$

For this case, the I/O's performed at the current level of recursion dominates, thus, $Q(N,P) = O\left(\frac{cN}{P}\right)$.
Case 2: Suppose $P \geq \frac{N}{B(B+1)}$.
Guess: $Q(N,P) = \frac{N}{P} + cB\log_{B+1} N$.

$$\begin{aligned} Q(N,P) &= Q\left(\frac{N}{B+1}, \frac{N}{B(B+1)}\right) + \frac{cN}{P} \\ &= \frac{cN}{B+1} \cdot \frac{B(B+1)}{N} + cB(\log_{B+1} N - 1) + \frac{cN}{P} \\ &= cB + cB\log_{B+1} N - cB + \frac{cN}{P} \\ &= cB\log_{B+1} N + \frac{cN}{P} \end{aligned}$$

Since $P \geq \frac{N}{B(B+1)}$, $Q(N,P) = O(B\log_{B+1} N)$. Additonally, we know that the recursion stops at a base case of size K, therefore we have $Q(N,P) = O(B\log_{B+1}\frac{N}{K})$.
Therefore, combining both cases results in $Q(N,P) = O\left(\frac{N}{P} + B\log_{B+1}\frac{N}{K}\right)$. □

Proposition 16.

$$\begin{aligned} Q(N,P) &= \begin{cases} O\left(N/B\right) & \text{if } N \leq M \text{ and } P = 1 \\ \left\lceil \frac{\sqrt{N}}{P} \right\rceil Q\left(\sqrt{N}, \left\lceil \frac{P}{\sqrt{N}} \right\rceil\right) + O\left(\frac{N}{P}\right) & \text{otherwise} \end{cases} \\ &= O\left(\frac{N}{P}\log\log_K N\right) . \end{aligned}$$

Proof. Guess: $Q(N,P) = \frac{cN}{P}\log\log N$.
Case 1: Suppose $P < \sqrt{N}$. Note that $\left\lceil \frac{P}{\sqrt{N}} \right\rceil = 1$.

$$\begin{aligned} Q(N,P) &= \frac{\sqrt{N}}{P} Q(\sqrt{N}, 1) + \frac{cN}{P} \\ &= \frac{\sqrt{N}}{P}(c\sqrt{N}\log\log\sqrt{N}) + \frac{cN}{P} \\ &= \frac{cN}{P}\log\left(\frac{1}{2}\log N\right) + \frac{cN}{P} \\ &= \frac{cN}{P}\log 2^{-1} + \frac{cN}{P}\log\log N + \frac{cN}{P} \\ &= -\frac{cN}{P} + \frac{cN}{P}\log\log N + \frac{cN}{P} \\ &= \frac{cN}{P}\log\log N . \end{aligned}$$

Case 2: Suppose $P \geq \sqrt{N}$. Note that $\left\lceil \frac{\sqrt{N}}{P} \right\rceil = 1$.

$$\begin{aligned} Q(N,P) &= Q\left(\sqrt{N}, \frac{P}{\sqrt{N}}\right) + \frac{cN}{P} \\ &= c\sqrt{N} \cdot \frac{\sqrt{N}}{P} \cdot \log\log\sqrt{N} + \frac{cN}{P} \\ &= \frac{cN}{P} \log\left(\frac{1}{2}\log N\right) + \frac{cN}{P} \\ &= \frac{cN}{P} \log 2^{-1} + \frac{cN}{P} \log\log N + \frac{cN}{P} \\ &= -\frac{cN}{P} + \frac{cN}{P} \log\log N + \frac{cN}{P} \\ &= \frac{cN}{P} \log\log N \ . \end{aligned}$$

Additonally, we know that the recursion stops at a base case of size K, therefore we have $Q(N,P) = O(\frac{N}{P} \log\log_K N)$. □

2.5.2 vEB Cycle-leader Algorithm

For the cycle-leader approach, we rely on performing parallel circular shifts of elements. We perform the circular shifts using the technique presented by Yang *et al.* [111], which involves two rounds of array reversals. We can reverse k elements in-place and in parallel by performing $\lfloor \frac{k}{2} \rfloor$ independent swaps. Specifically, index i swaps with index $k - i - 1$ (using 0-indexing). Thus, to optimize for I/Os, we can swap elements in groups of B, provided that every group of B elements are located in contiguous memory locations. Therefore, we can perform a circular shift of N elements in $O(\frac{N}{PB})$ I/Os. For the remainder of the section, assume every circular shift uses this optimization.

The vEB cycle-leader approach, described in Section 2.4.1, employs the equidistant gather operation, which relies on circular shifts. However, the equidistant gather performs a circular shift on elements strided by distance $O(\sqrt{N})$ and are thus not in contiguous memory. To avoid the I/O inefficiency of such an access pattern, we propose an initial transposition phase to block elements in each disjoint cycle together.

We can view the sorted array as an $(r+1) \times (l+1)$ row-major matrix with the bottom-right element removed. We can ignore the last row, which contains $\mathcal{T}_{r+1}$, since these elements do not move during the cycles. We also ignore the last column of the matrix, which contains the r elements of $\mathcal{T}_0$. Additionally for $r < l$, we ignore the remaining right-most $(l-r)$ columns, as these elements do not participate in any cycles.

Thus, we consider a square matrix of size $r \times r$. Figure 2.5 illustrates this representation for $r = l$, how each subtree is contained therein, and what elements are contained in each disjoint cycle of the gather. To improve I/O efficiency, we perform a circular shift on each row i by i positions

to the right, which aligns the elements in each disjoint cycle into columns. We then transpose the square matrix to align the elements in each cycle into rows, placing all elements of each cycle into contiguous memory.

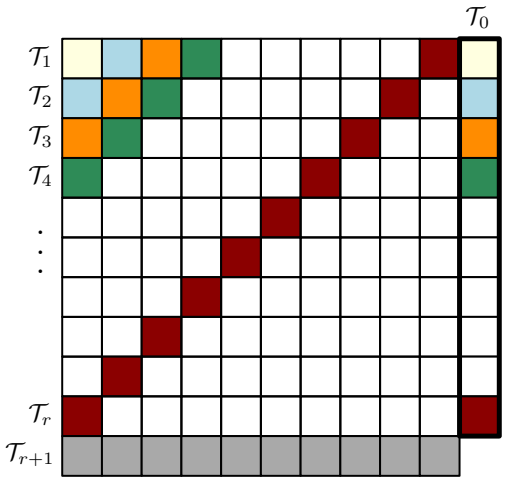

Figure 2.5: Illustration of the distinct cycles of the equidistant gather operation for $r = l$. We consider memory as an $(r+1) \times (l+1)$ matrix. Shifting each row and transposing the inner $r \times r$ matrix lets us perform each cycle I/O efficiently.

Shifting r rows of r elements requires $O(\frac{r^2}{PB})$ I/Os. Assuming that $P \leq N/B^2$ and $M \geq B^2$, we can perform matrix transposition in $O(\frac{r^2}{PB})$ I/Os by tiling the matrix into sub-matrices of size $B \times B$ [2, 104]. With each disjoint cycle in contiguous memory, we can now permute the first set of cycles of the equidistant gather I/O efficiently and in parallel. Thus for $r = O(\sqrt{N})$, this permutation takes $\frac{1}{P}\sum_{i=1}^{r}\left(1 + O\left(\frac{i}{B}\right)\right) = O\left(\frac{\sqrt{N}}{P} + \frac{N}{PB}\right) = O\left(\frac{N}{PB}\right)$ I/Os, assuming that $B \leq \sqrt{N}$.

After performing the first set of disjoint cycles, we perform the inverse of the above transposition to permute the elements back into their original order (this places each $\mathcal{T}_i$ into contiguous memory). To do this, we transpose the $r \times r$ matrix and perform a left circular shift on each row i by i positions. We complete the equidistant gather operation by performing a left circular shift on each subtree $\mathcal{T}_i$ by $i - 1$ positions. As outlined in Section 2.4.2, the equidistant gather operation is applied recursively to perform the vEB layout permutation. The I/O complexity of this algorithm is:

Proposition 17.

$$\begin{aligned} Q(N,P) &= \begin{cases} O(N/B) & \text{if } N \leq M \text{ and } P = 1 \\ \left\lceil \frac{\sqrt{N}}{P} \right\rceil Q\left(\sqrt{N}, \left\lceil \frac{P}{\sqrt{N}} \right\rceil\right) + O\left(\frac{N}{PB}\right) & \text{otherwise} \end{cases} \\ &= O\left(\frac{N}{PB} \log\log_K N\right) . \end{aligned}$$

Proof. Guess: $Q(N,P) = \frac{cN}{PB} \log\log N$.

Case 1: Suppose $P < \sqrt{N}$. Note that $\left\lceil \frac{P}{\sqrt{N}} \right\rceil = 1$.

$$\begin{aligned}
Q(N,P) &= \frac{\sqrt{N}}{P} Q(\sqrt{N}, 1) + \frac{cN}{PB} \\
&= \frac{\sqrt{N}}{P} \left(c \frac{\sqrt{N}}{B} \log\log \sqrt{N} \right) + \frac{cN}{PB} \\
&= \frac{cN}{PB} \log \left(\frac{1}{2} \log N \right) + \frac{cN}{PB} \\
&= \frac{cN}{PB} \log 2^{-1} + \frac{cN}{PB} \log\log N + \frac{cN}{PB} \\
&= -\frac{cN}{PB} + \frac{cN}{PB} \log\log N + \frac{cN}{PB} \\
&= \frac{cN}{PB} \log\log N \ .
\end{aligned}$$

Case 2: Suppose $P \geq \sqrt{N}$. Note that $\left\lceil \frac{\sqrt{N}}{P} \right\rceil = 1$.

$$\begin{aligned}
Q(N,P) &= Q\left(\sqrt{N}, \frac{P}{\sqrt{N}} \right) + \frac{cN}{PB} \\
&= c \frac{\sqrt{N}}{B} \cdot \frac{\sqrt{N}}{P} \cdot \log\log \sqrt{N} + \frac{cN}{PB} \\
&= \frac{cN}{PB} \log \left(\frac{1}{2} \log N \right) + \frac{cN}{PB} \\
&= \frac{cN}{PB} \log 2^{-1} + \frac{cN}{PB} \log\log N + \frac{cN}{PB} \\
&= -\frac{cN}{PB} + \frac{cN}{PB} \log\log N + \frac{cN}{PB} \\
&= \frac{cN}{PB} \log\log N \ .
\end{aligned}$$

Additonally, we know that the recursion stops at a base case of size K, therefore we have $Q(N,P) = O(\frac{N}{PB} \log\log_K N)$. □

Alternatively, a simpler solution would be to forgo the above described transposition phase and assign each processor a group of $O(B)$ cycles to permute sequentially. This can be done I/O efficiently since B consecutive elements will always contain elements from the same B cycles. The resulting I/O complexity is $O\left(\left(\frac{N}{PB} + \frac{\sqrt{N}}{B} \right) \log\log_K N \right)$. Although not as asymptotically efficient for large values of P, in practice for most architectures $P \leq \sqrt{N}$ and the first term will dominate, resulting in the same asymptotic complexity.

2.5.3 B-tree Cycle-leader Algorithm

Recall from Section 2.4.2 that the B-tree cycle-leader algorithm is recursive, performing the equidistant gather operation while considering chunks of C elements as single units. Thus, as long as $C \geq B$, every swap of C elements will be I/O-efficient. Since $C = \left\lceil \frac{N}{(B+1)^2} \right\rceil$ for $N = (B+1)^{h+1} - 1$, only the base case ($C = 1$) will have a chunk size less than B. However, assuming that $M \geq (B+1)^2 - 1 = \Theta(B^2)$, we can simply load the base case into internal memory to perform the permutation in $O(B)$ I/Os. All other recursive levels are performed I/O efficiently, thus:

Proposition 18.

$$\begin{aligned} Q'(N,P) &= \begin{cases} O(N/B) & \text{if } N \leq M \text{ and } P = 1 \\ \left\lceil \frac{B+1}{P} \right\rceil Q'\left(\frac{N}{B+1}, \left\lceil \frac{P}{B+1} \right\rceil\right) + O\left(\frac{N}{PB}\right) & \text{otherwise} \end{cases} \\ &= O\left(\frac{N}{PB} \log_{B+1} \frac{N}{K}\right) . \end{aligned}$$

Proof. Guess: $Q'(N,P) = \frac{cN}{PB} \log_{B+1} N$.
Case 1: Suppose $P < B+1$. Note that $\left\lceil \frac{P}{B+1} \right\rceil = 1$.

$$\begin{aligned} Q'(N,P) &= \frac{B+1}{P} \cdot Q'\left(\frac{N}{B+1}, 1\right) + \frac{cN}{PB} \\ &= \frac{B+1}{P}\left(\frac{cN}{B(B+1)}\left(\log_{B+1} N - 1\right)\right) + \frac{cN}{PB} \\ &= \frac{cN}{PB} \log_{B+1} N - \frac{cN}{PB} + \frac{cN}{PB} \\ &= \frac{cN}{PB} \log_{B+1} N . \end{aligned}$$

Case 2: Suppose $P \geq B+1$. Note that $\left\lceil \frac{B+1}{P} \right\rceil = 1$.

$$\begin{aligned} Q'(N,P) &= Q'\left(\frac{N}{B+1}, \frac{P}{B+1}\right) + \frac{cN}{PB} \\ &= \frac{cN}{B(B+1)} \cdot \frac{B+1}{P}\left(\log_{B+1} N - 1\right) + \frac{cN}{PB} \\ &= \frac{cN}{PB} \log_{B+1} N - \frac{cN}{PB} + \frac{cN}{PB} \\ &= \frac{cN}{PB} \log_{B+1} N . \end{aligned}$$

Additonally, we know that the recursion stops at a base case of size K, therefore $Q'(N,P) = O(\frac{N}{PB} \log_{B+1} \frac{N}{K})$. □

Proposition 19.

$$Q(N,P) = \begin{cases} O(N/B) & \text{if } N \le M \text{ and } P = 1 \\ Q\left(\frac{N}{B+1}, \text{MIN}(P, \frac{N}{B(B+1)})\right) + Q'(N,P) & \text{otherwise} \end{cases}$$
$$= O\left(\left(\frac{N}{PB} + \log_{B+1}\frac{N}{K}\right)\log_{B+1}\frac{N}{K}\right) .$$

Proof. Let $h = \log_{B+1} N$.
Case 1: Suppose $P < \frac{N}{N(B+1)}$.
Guess: $Q(N,P) = \left(\frac{B+1}{B}\right)\frac{cN}{PB}h - \left(\frac{B+1}{B^2}\right)\frac{cN}{PB}$.

$$\begin{aligned} Q(N,P) &= Q\left(\frac{N}{B+1}, P\right) + \frac{cN}{PB}h \\ &= \left(\frac{B+1}{B}\right)\frac{cN}{PB(B+1)}(h-1) - \left(\frac{B+1}{B^2}\right)\frac{cN}{PB(B+1)} + \frac{cN}{PB}h \\ &= \left(\frac{1}{B}\right)\frac{cN}{PB}h - \left(\frac{1}{B}\right)\frac{cN}{PB} - \left(\frac{1}{B^2}\right)\frac{cN}{PB} + \frac{cN}{PB}h \\ &= \left(\frac{1}{B}+1\right)\frac{cN}{PB}h - \left(\frac{1}{B}+\frac{1}{B^2}\right)\frac{cN}{PB} \\ &= \left(\frac{1}{B}+\frac{B}{B}\right)\frac{cN}{PB}h - \left(\frac{B}{B^2}+\frac{1}{B^2}\right)\frac{cN}{PB} \\ &= \left(\frac{B+1}{B}\right)\frac{cN}{PB}h - \left(\frac{B+1}{B^2}\right)\frac{cN}{PB} . \end{aligned}$$

Case 2: Suppose $P \ge \frac{N}{N(B+1)}$.
Guess: $Q(N,P) = \frac{cN}{PB}h + \frac{c}{2}h^2 - \frac{c}{2}h$.

$$\begin{aligned} Q(N,P) &= Q\left(\frac{N}{B+1}, \frac{N}{B(B+1)}\right) + \frac{cN}{PB}h \\ &= \frac{cN}{B(B+1)}\cdot\frac{B(B+1)}{N}\cdot(h-1) + \frac{c}{2}(h-1)^2 - \frac{c}{2}(h-1) + \frac{cN}{PB}h \\ &= ch - c + \frac{c}{2}(h^2 - 2h + 1) - \frac{c}{2}h + \frac{c}{2} + \frac{cN}{PB}h \\ &= \frac{c}{2}h - \frac{c}{2} + \frac{c}{2}h^2 - ch + \frac{c}{2} + \frac{cN}{PB}h \\ &= \frac{c}{2}h^2 - \frac{c}{2}h + \frac{cN}{PB}h . \end{aligned}$$

Since $P \ge \frac{N}{N(B+1)}$, this case simplifies to $Q(N,P) = O(\log^2_{B+1} N)$.
Combining both cases results in $Q(N,P) = O\left(\left(\frac{N}{PB} + \log_{B+1} N\right)\log_{B+1} N\right)$. Additionally, we know that the recursion stops at a base case of size K, therefore $Q(N,P) = O\left(\left(\frac{N}{PB} + \log_{B+1}\frac{N}{K}\right)\log_{B+1}\frac{N}{K}\right)$.
□

Recall from Section 2.4.3 that the BST algorithm is a special case of the B-tree algorithm where the node size is a single element. In this case, the last $\Theta(\log B)$ rounds will have a chunk size less than B. Thus, once $N = O(B)$, we load the array into internal memory which results in $O\left(\left(\frac{N}{PB} + \log \frac{N}{K}\right) \log \frac{N}{K}\right)$ I/Os.

2.6 Extensions to non-perfect trees

Since the array is given in sorted order, any arbitrary size BST or B-tree will be complete (though not necessarily perfect). Hence for BSTs and B-trees, we can first permute the non-full level of leaves to the end of the array. For a tree of height h, the number of *full elements*, i.e., the elements in the full levels, in a BST is $I = 2^h - 1$, and in a B-tree $I = (B+1)^h - 1$. The number of *non-full elements*, i.e., the elements in the non-full level, is $L = N - I$. In both trees, the parents of non-full elements are initially located in the array such that they partition the non-full elements. We gather them to the front of the array and shift the non-full elements to the end of the array via a circular shift. To perform this gather, we apply a $(B+1)$-way un-shuffle (and additionally a B-way shuffle on the non-full elements for B-trees) as seen in Section 2.3.2. Alternatively, we can apply the extended equidistant gather operation described in Section 2.4.2. This process takes $D(N) = O(T_{REV_2}(L))$ depth and $W(N) = O(L \cdot T_{REV_2}(L) + N)$ work for BSTs (via 2-way un-shuffle); and $D(N) = O(\log_{B+1} L)$ depth and $W(N) = O(L(B+1) \cdot \log_{B+1} L + N)$ work for B-trees (via extended equidistant gather). After applying this initial stage, we can proceed with the algorithm on the full elements which form a perfect tree of height $h - 1$.

Recall from Section 2.1.1 that $r = 2^{\lfloor (h-1)/2 \rfloor} - 1$ is the size of the top subtree, $\mathcal{T}_0$, and $l = 2^{\lceil (h-1)/2 \rceil} - 1$ is the size of each of the first $y = \lfloor (N-r)/l \rfloor$ bottom subtrees, $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_y$. We first gather the first y elements of $\mathcal{T}_0$, which are initially located at every $(l+1)$-th array location, to the front of the array. Then we shift the remaining $r - y$ elements of $\mathcal{T}_0$, which reside at the end of the array, to the front of the array after the first y elements of $\mathcal{T}_0$. To perform this gather, we can either perform the equidistant gather operation described in Section 2.4.1; or we can apply an $(l+1)$-way un-shuffle followed by an l-way shuffle on the elements in $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_y$. The resulting work and depth of this process using the equidistant gather is $W(N) = O(N)$ and $D(N) = O(1)$. After this initial permutation, we recurse on subtrees $\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_y$ using the algorithm for the perfect vEB layout. If $x = \lceil (N-r)/l \rceil = y + 1$, then we additionally recurse on the last non-perfect subtree $\mathcal{T}_x$ using the algorithm just described for the non-perfect vEB layout.

2.7 Experimental Optimizations

2.7.1 Query Optimization

Brodal et al. [14] describe a vEB query approach that utilizes a precomputed table of size $O(\log N)$. Consider an arbitrary node in the vEB located at depth d and unfold the vEB recursion such that the considered node is the root of a vEB bottom tree. The precomputed table stores at index d the number of nodes in the corresponding bottom tree, the number of nodes in the corresponding top tree, and the depth of the root of the corresponding top tree. Thus, when performing a vEB query, the precomputed table can be used to calculate the index of the next node in $O(1)$ time. For non-perfect vEBs, the last leaf subtree may contain fewer elements than other leaf subtrees and thus, may have a different subtree height. Due to this, an additional table is needed to be able to query the non-perfect leaf subtree. This is needed for all non-perfect leaf subtrees for all recursive levels of the vEB. In the worst case, this requires $\sum_{i=0}^{\log\log N} \frac{\log N}{2^i} = O(\log N)$ space. This query optimization using the precomputed table(s) is used throughout Section 2.8.

2.7.2 Hybrid BST Layout

For permuting non-perfect BSTs, we must perform an initial permutation to gather and shift the non-full elements to the end of the array (as described in Section 2.6). To do this, we can either use the equidistant gather operation (i.e., cycle-leader approach) or a 2-way un-shuffle (i.e., involution approach).

In the conference version of this work, we found that the involution approach performs better than the cycle-leader approach for perfect BSTs. However, recall from Section 2.3.1 that the BST involution approach uses a pair of involutions to permute the sorted array into the BST layout, which does not require the use of any shuffles or un-shuffles. In comparison, the involution approaches that do use shuffles and un-shuffles (e.g. the B-tree involution permutation) do not perform well. Therefore, we additionally consider the BST hybrid approach, which uses the extended equidistant gather for the initial permutation to gather the non-full elements, then uses the pair of involutions to permute the full elements into the BST layout. The BST hybrid approach is additionally used in Section 2.8.

2.7.3 Modified van Emde Boas Layout

In the conference version of this work, results on the vEB permutation algorithms showed poor performance on GPUs [10]. This slowdown is attributed to the recursive implementation of these algorithms, which is known to degrade performance on GPUs. However, developing iterative versions of these algorithms on the GPU is challenging, due to the potentially uneven size of the top and bottom trees in the vEB layout. Instead, we define a variant of the vEB layout, which we call the *modified van Emde Boas* layout (mvEB).

In the mvEB layout, the height of the bottom trees are rounded up to the nearest power of two (at the expense of the top subtree's height being shortened by the same change in height as the leaf subtrees). In this way, the bottom subtrees are guaranteed to always be perfectly balanced, i.e., all of the top and bottom trees in the following recursive divisions always contain the same number of nodes. This makes developing an iterative version for the perfectly balanced subtrees possible in a single GPU kernel, for each recursive division. In Section 2.8.2, the mvEB layout is used instead of the recursively implemented vEB layout.

2.8 Experiments

2.8.1 Methodology

We evaluate the performance of our search tree permutation algorithms on 3 NVIDIA GPUS: (1) a NVIDIA Tesla K40 with 2,880 compute cores, 12 GB of GDDR5 type global memory with a theoretical bandwidth of 288 GB/s, and compute capability 3.5; (2) a NVIDIA Quadro M4000 with 1,664 compute cores, 8 GB of GDDR5 type global memory with a theoretical bandwidth of 192 GB/s, and compute capability 5.2; and (3) a NVIDIA GeForce RTX 2080 Ti with 4,352 compute cores, 11 GB of GDDR6 type global memory with a theoretical bandwidth of 616 GB/s, and compute capability 7.5. We use the CUDA 10.1 compiler [84] with the `-O3` and `-use_fast_math` flags.

Experiments are conducted on arrays of 32-bit integer values ($B = 32$) on our GPU platforms. Permutation experiments are conducted on both perfect trees (powers of 2 minus 1 for BSTs and vEBs; and powers of $(B + 1)$ minus 1 for B-trees) and non-perfect trees (powers of 10 for all trees; and additionally powers of 2 minus 1 for B-trees). All experimental results are averages over 10 trials and queries are randomly sampled from a uniform distribution of $1, 2, \ldots n$, making all the searches 100% successful. The code used in these experiments can be found at: `https://github.com/algoparc/Tree-Layouts`.

2.8.2 Results

Two key features of the GPU architecture make it compelling for this work: (1) GPUs have a relatively small memory, making the in-place feature of our permutation algorithms crucial; and (2) GPUs have high memory throughput and many compute cores, making them effective for a large number of independent search queries. We note that when accessing global memory, coalesced accesses are recommended to minimize the number of memory transactions. Thus by using $B = 32$, each thread in a warp will accesses consecutive memory locations resulting in $O(1)$ global memory transactions.

Recall that we use our modified van Emde Boas layout (Section 2.7.3), rather than the van Emde Boas layout. While each of our algorithms provides a high degree of parallelism, synchronization

and communication overheads can significantly degrade GPU performance. Due to this reason, we assign each thread to a query and have threads execute independently of each other. For the B-tree layout, in order to access each node of $B = 32$ elements in a coalesced manner, each warp is assigned to a query and warp-level communication primitives are utilized to coordinate the search.

Figure 2.6 (respectively Figure 2.7 and Figure 2.8) plots the average permutation time versus the input size, for the K40 (respectively Quadro M4000 and RTX 2080 Ti). For all three GPU platforms, the general consensus is that the modified van Emde Boas cycle-leader approach is the fastest permutation algorithm. This is expected since the vEB algorithm has the lowest I/O complexity. On the K40 and Quadro M4000 GPUs, both the B-tree cycle-leader and BST involution approaches are competitive until the BST involution algorithm shows a sharp increase in runtime for $N > 2^{29} - 1$ elements. Similar to our CPU platform, the BST involutions and BST hybrid permutations show a significant runtime increase for non-perfect trees.

Figure 2.9 (respectively Figure 2.11 and Figure 2.13) shows the average time to perform 1 million queries on each search tree layout and binary search on a sorted array versus the input size, for the K40 (respectively Quadro M4000 and RTX 2080 Ti). It is interesting to see that the query performance varies depending on the GPU platform used. On all GPU platforms, B-tree querying results in the fastest runtime for $N > 2^{28} - 1$ on the K40 and Quadro M400 and $N > 2^{23} - 1$ on the RTX 2080 Ti. On the K40 and Quadro M4000, BST querying outperforms mvEB querying; however on the RTX 2080 Ti, mvEB querying outperforms BST querying for $N \geq 2^{24} - 1$. Furthermore, on the K40 and Quadro M4000 GPUs, a large increase in runtime is observed for $N > 2^{29} - 1$. For this reason, we measure the total runtime of permuting and performing Q queries on one perfect tree and one non-perfect tree for $N \leq 2^{29} - 1$ and similarly for $N > 2^{29} - 1$.

Figure 2.15, Figure 2.16, and Figure 2.17 shows the combined runtime of permuting and querying each search tree layout with $N = 100$ million elements on the K40, Quadro M4000, and RTX 2080 Ti, respectively. Figure 2.18, Figure 2.19, and Figure 2.20 shows the combined runtime of permuting and querying each search tree layout with $N = 2^{29} - 1$ elements on the K40, Quadro M4000, and RTX 2080 Ti, respectively. Figure 2.21, Figure 2.22, and Figure 2.23 shows the combined runtime of permuting and querying each search tree layout with $N = 100$ million elements on the K40, Quadro M4000, and RTX 2080 Ti, respectively. And lastly, Figure 2.24, Figure 2.25, and Figure 2.26 plot the combined runtime for $N = 2^{30} - 1$ elements on the K40, Quadro M4000, and RTX 2080 Ti, respectively. Table 2.8.2 summarizes the number of queries for which it becomes worthwhile to permute and query on the respective search tree layout (compared to an equal number of binary search queries). For non-perfect trees ($N = 100$ million and $N = 1$ billion), we see a significant increase in the percentage of queries needed for the BST layout, mainly due to the high cost of permutation. For $N = 100$ million on the K40, the mvEB layout is never worth the cost of permuting because binary search is faster than querying the mvEB layout until $N \geq 2^{27} - 1$. Overall, for all input sizes, the B-tree layout results in the lowest number of queries needed on

the K40 and Quadro M4000 GPUs, as it has both the fastest permutation and query runtimes. However, on the RTX 2080 Ti, the mvEB is the best performing layout, due to it having the fastest permutation runtime and second fastest query runtime. We note that for a large number of queries on the RTX 2080 Ti, we expect the better query performance of the B-tree layout to eventually overcome the faster permutation runtime of the mvEB layout.

Layout	K40	Quadro M4000	RTX 2080 Ti
BST	34 million (34% of N)	47 million (47% of N)	83 million (83% of N)
B-tree	13 million (13% of N)	20 million (20% of N)	23 million (23% of N)
mvEB	—	44 million (44% of N)	13 million (13% of N)

Layout	K40	Quadro M4000	RTX 2080 Ti
BST	62 million (11.55% of N)	65 million (12.11% of N)	293 million (54.58% of N)
B-tree	45 million (8.38% of N)	55 million (10.24% of N)	119 million (22.17% of N)
mvEB	127 million (23.66% of N)	106 million (19.74% of N)	50 million (9.31% of N)

Layout	K40	Quadro M4000	RTX 2080 Ti
BST	71 million (7.1% of N)	53 million (5.3% of N)	936 million (93.6% of N)
B-tree	14 million (1.4% of N)	14 million (1.4% of N)	194 million (19.4% of N)
mvEB	20 million (2% of N)	32 million (3.2% of N)	103 million (10.3% of N)

Layout	K40	Quadro M4000	RTX 2080 Ti
BST	62 million (5.77% of N)	62 million (5.77% of N)	596 million (55.51% of N)
B-tree	14 million (1.3% of N)	15 million (1.4% of N)	217 million (20.21% of N)
mvEB	19 million (1.77% of N)	32 million (2.98% of N)	96 million (8.94% of N)

Table 2.2: Number of queries needed for it to be beneficial (compared to an equal number of binary search queries) to perform each of the search tree layout permutations on each of our GPU platforms with $N = 100$ million (first table), $N = 2^{29} - 1$ (second table), $N = 1$ billion (third table), and $N = 2^{30} - 1$ (fourth table). On the K40 and Quadro M4000, the B-tree layout has the lowest number of queries needed; while on the RTX 2080 Ti, the modified van Emde Boas (mvEB) layout beats both the B-tree and BST layouts.

2.9 Conclusion

Implicit search tree layouts can improve search query performance by exploiting locality of reference and, consequently, cache efficiency. However, given initially sorted input, permuting it into a search tree layout requires extra space and can be costly, thereby bringing into question the usefulness of

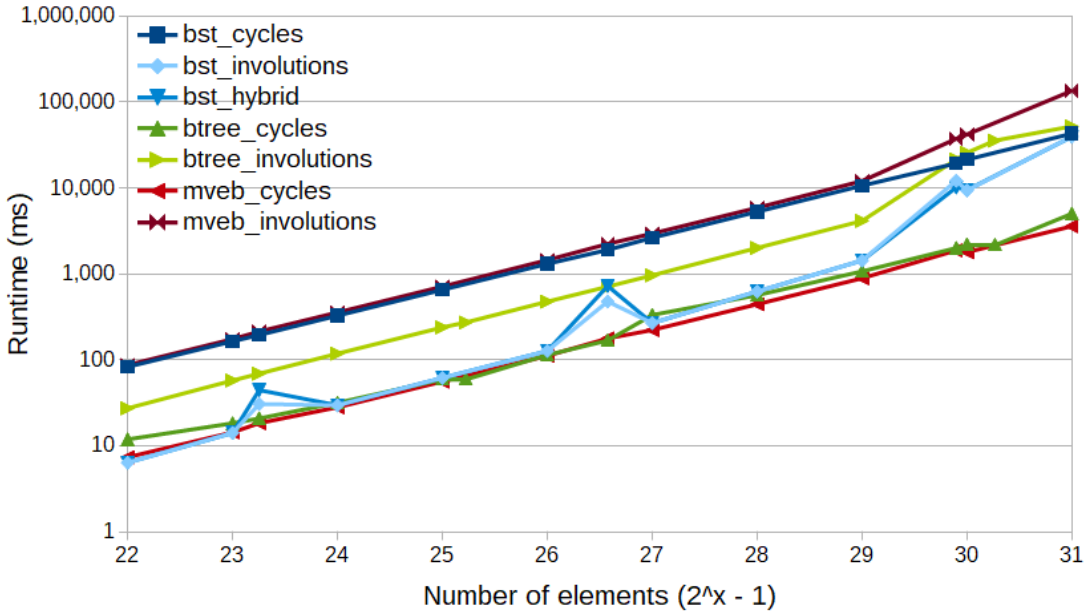

Figure 2.6: Average time to permute a sorted array using each permutation algorithm on the NVIDIA K40. The graph is displayed on a log-log scale.

implicit search tree layouts in memory-constrained environments and/or when few search queries need to be performed.

In this chapter we present parallel in-place algorithms for permuting a sorted array into popular search tree layouts. Our algorithms exhibit the following features which make them exceptionally practical: 1) they operate in-place, making it possible to permute inputs that occupy all available space; 2) they are efficient in parallel, allowing the use of many-core architectures; and 3) our cycle-leader algorithms are I/O-efficient, resulting in implementations that utilize the cache hierarchy effectively. This work underscores the importance of I/O-efficiency when designing parallel algorithms for modern manycore hardware, such as GPUs. The development of efficient memory layouts, beyond searching, provides fertile ground for future research.

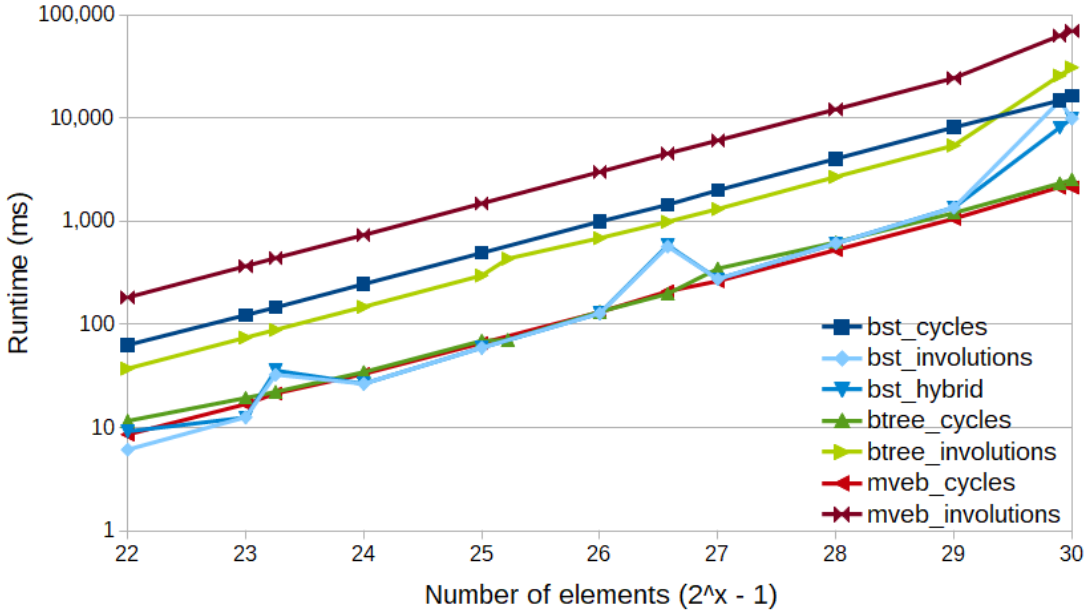

Figure 2.7: Average time to permute a sorted array using each permutation algorithm on the NVIDIA Quadro M4000. The graph is displayed on a log-log scale.

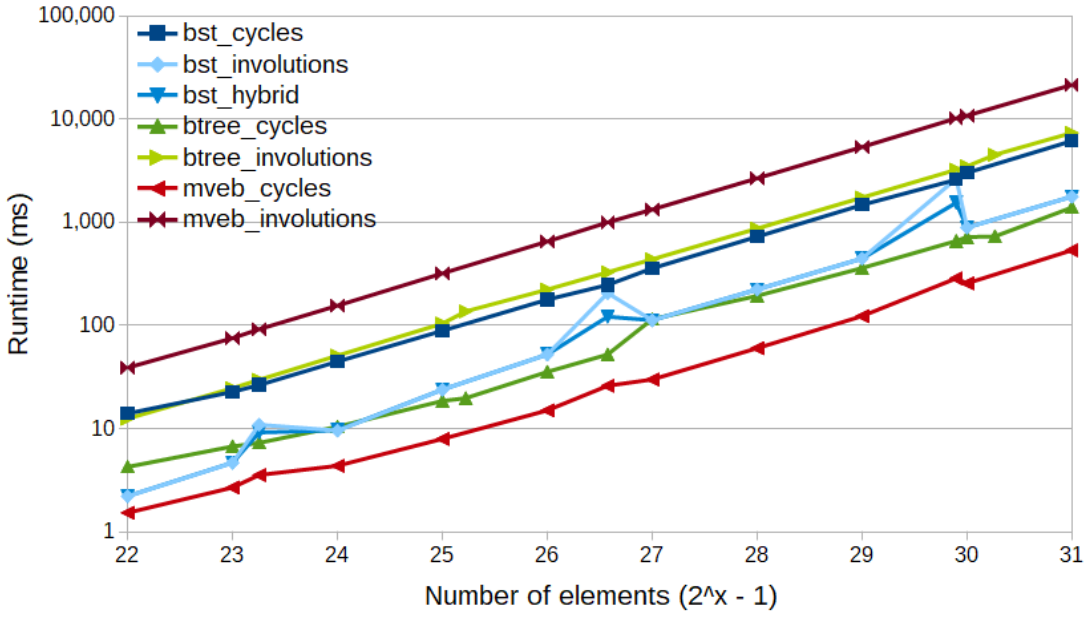

Figure 2.8: Average time to permute a sorted array using each permutation algorithm on the NVIDIA GeForce RTX 2080 Ti. The graph is displayed on a log-log scale.

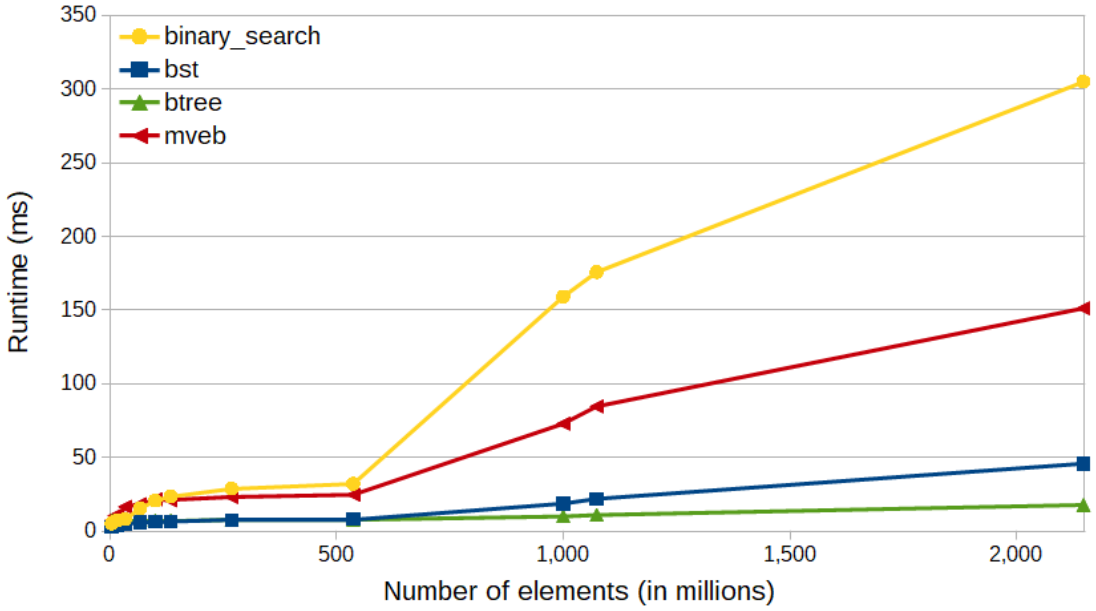

Figure 2.9: Average time to perform 1 million queries on each search tree layout and binary search on a sorted array on the NVIDIA K40. The graph is displayed on a linear scale to emphasize the logarithmic shape of the querying.

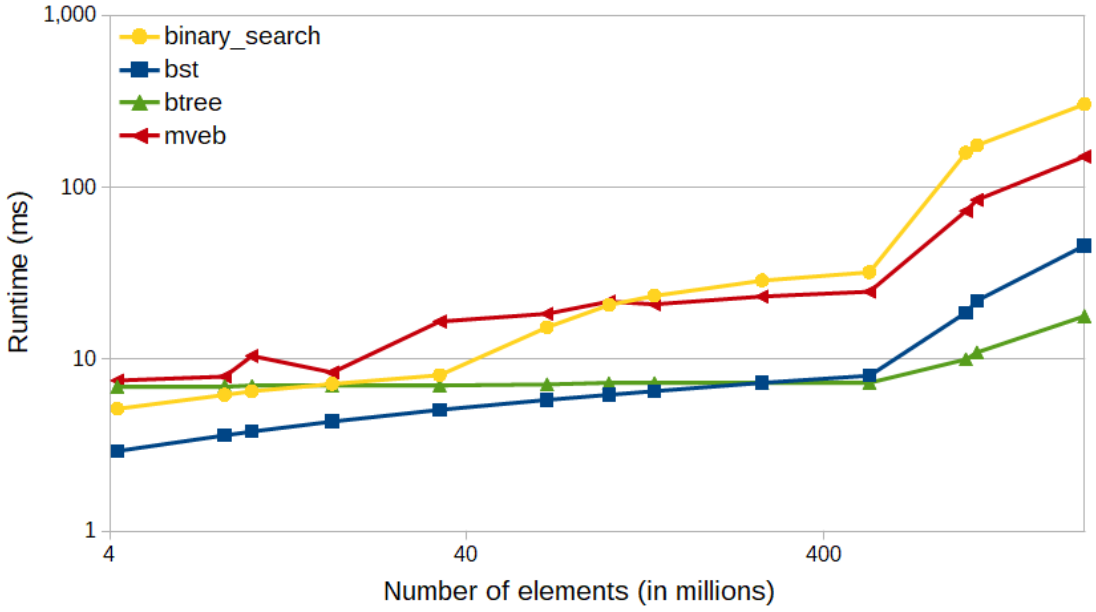

Figure 2.10: Average time to perform 1 million queries on each search tree layout and binary search on a sorted array on the NVIDIA K40. The graph is displayed on a log-log scale to see the comparison of results between different array layouts.

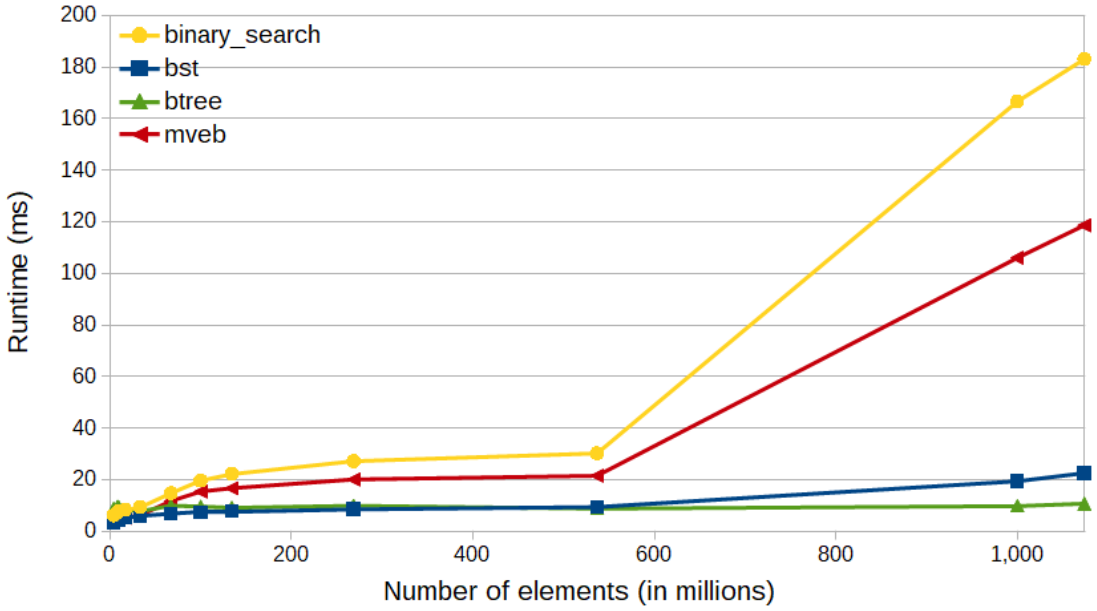

Figure 2.11: Average time to perform 1 million queries on each search tree layout and binary search on a sorted array on the NVIDIA Quadro M4000. The graph is displayed on a linear scale to emphasize the logarithmic shape of the querying.

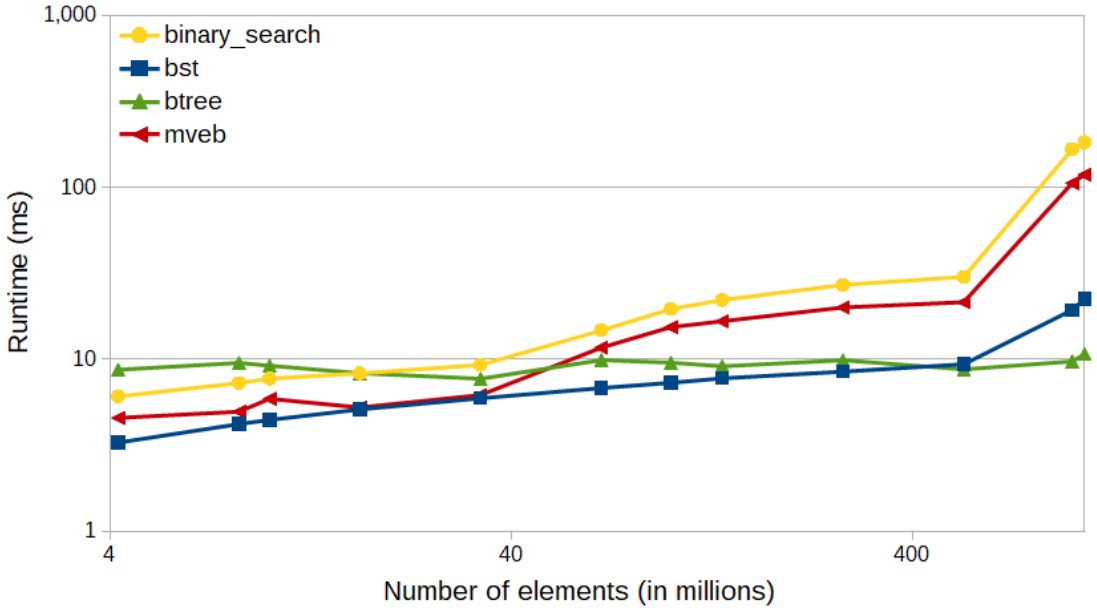

Figure 2.12: Average time to perform 1 million queries on each search tree layout and binary search on a sorted array on the NVIDIA Quadro M4000. The graph is displayed on a log-log scale to see the comparison of results between different array layouts.

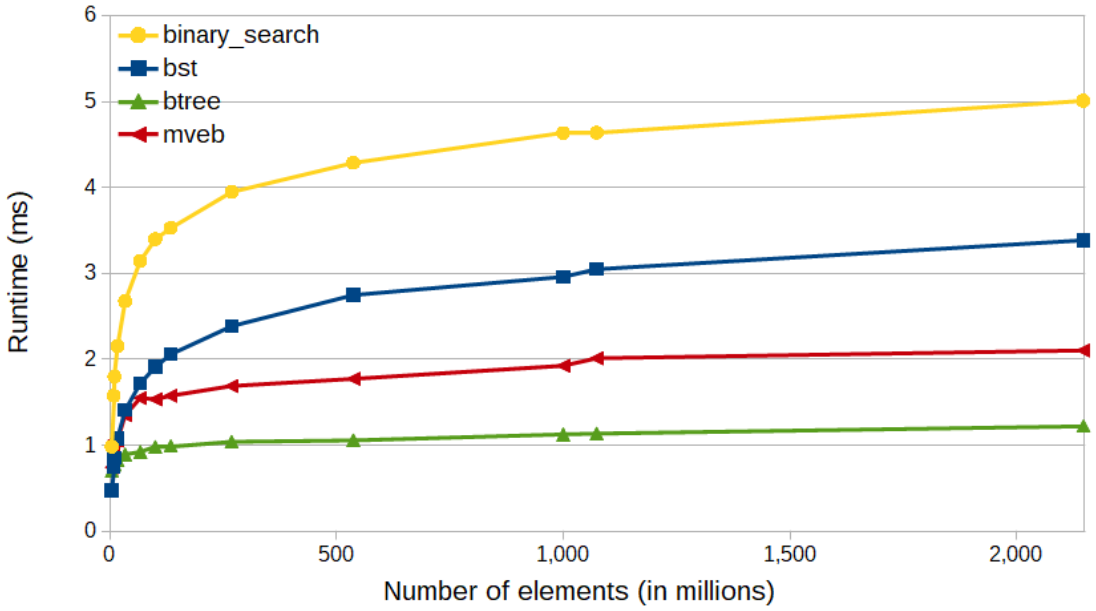

Figure 2.13: Average time to perform 1 million queries on each search tree layout and binary search on a sorted array on the NVIDIA GeForce RTX 2080 Ti. The graph is displayed on a linear scale to emphasize the logarithmic shape of the querying.

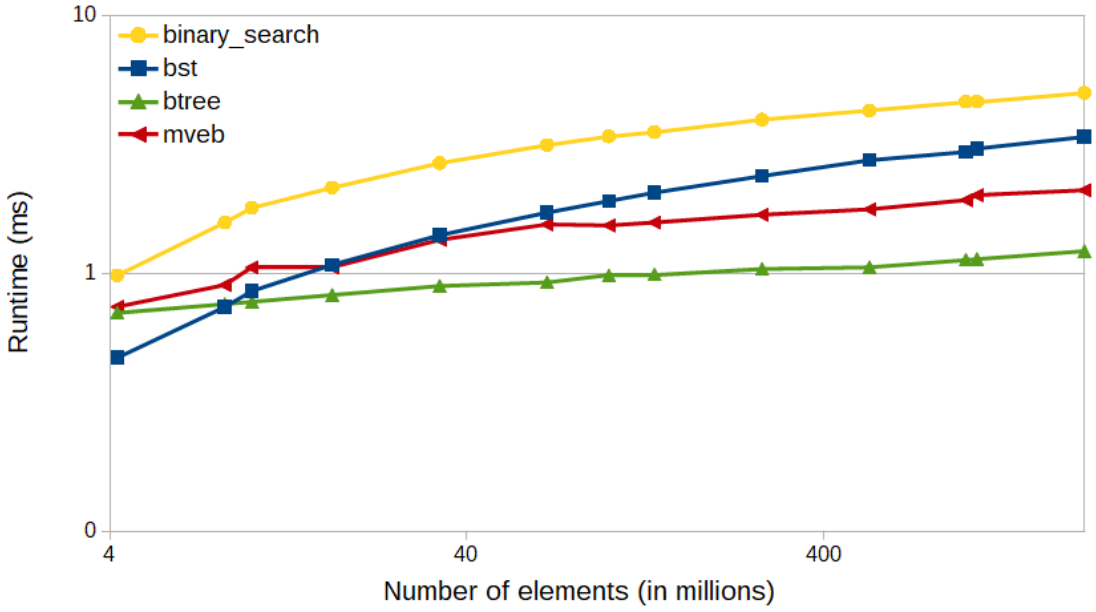

Figure 2.14: Average time to perform 1 million queries on each search tree layout and binary search on a sorted array on the NVIDIA GeForce RTX 2080 Ti. The graph is displayed on a log-log scale to see the comparison of results between different array layouts.

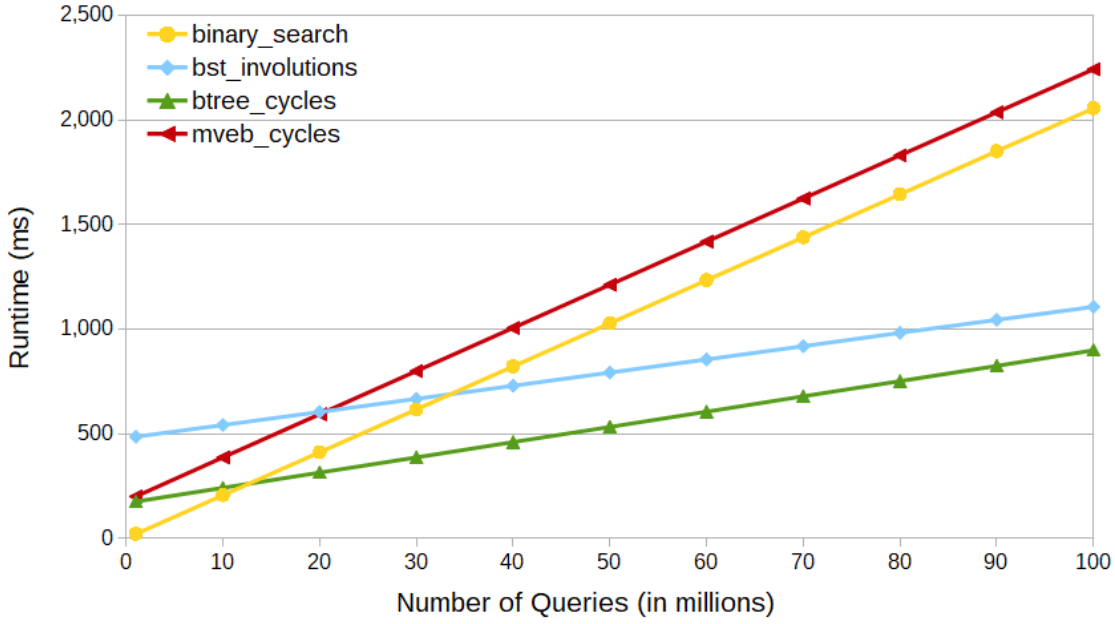

Figure 2.15: Combined time to permute and query each layout on the NVIDIA K40 with $N = 100$ million elements. The fastest permutation algorithm is used for each tree layout.

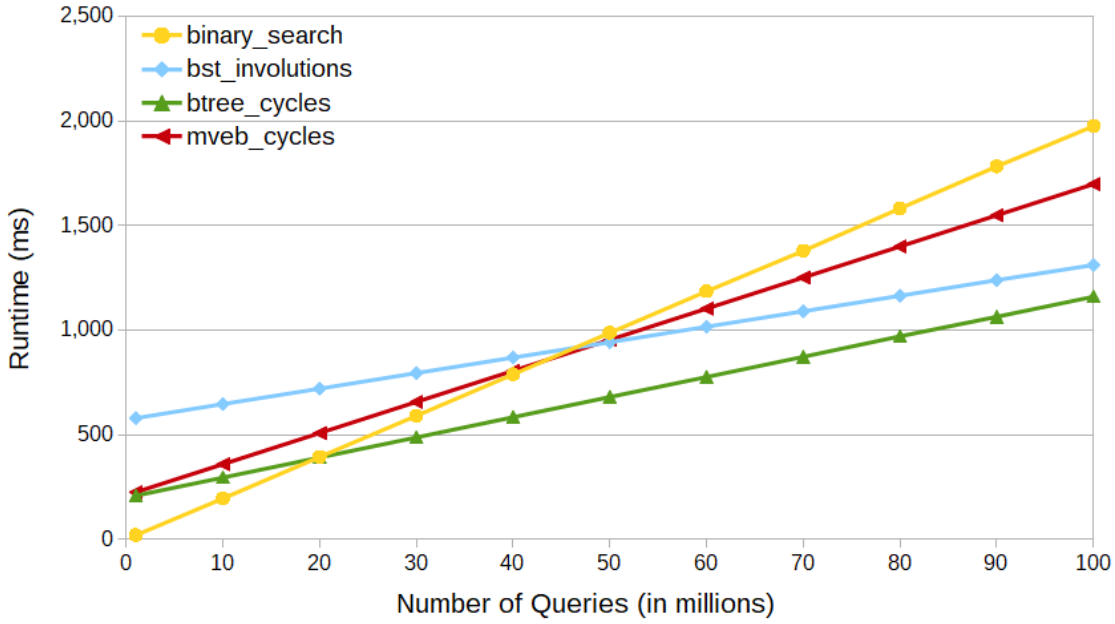

Figure 2.16: Combined time to permute and query each layout on the NVIDIA Quadro M4000 with $N = 100$ million elements. The fastest permutation algorithm is used for each tree layout.

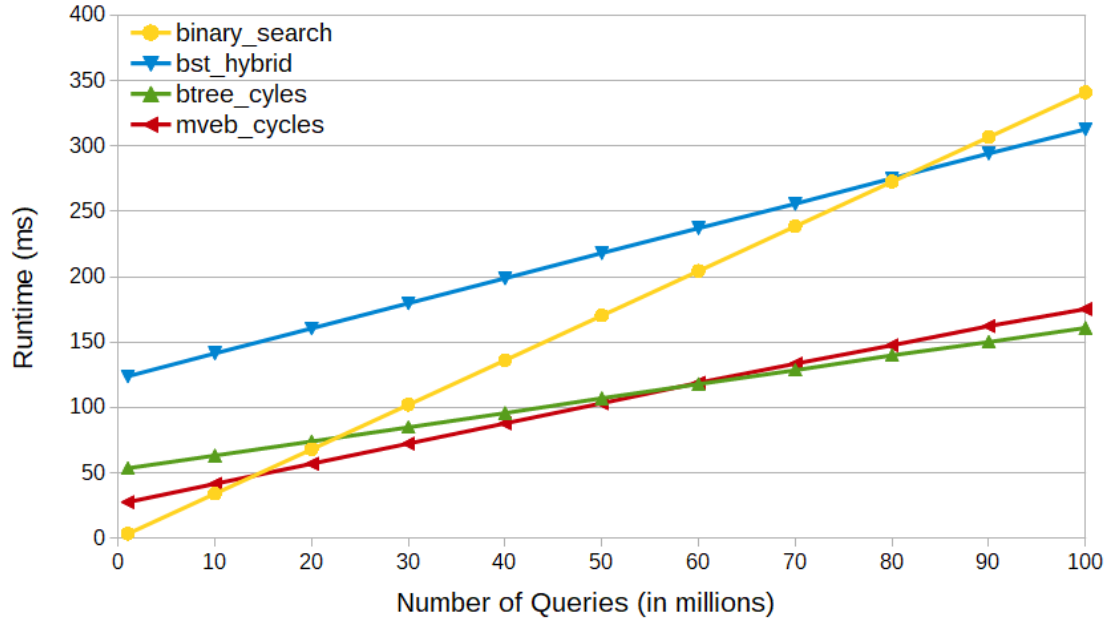

Figure 2.17: Combined time to permute and query each layout on the NVIDIA RTX 2080 Ti with $N = 100$ million elements. The fastest permutation algorithm is used for each tree layout.

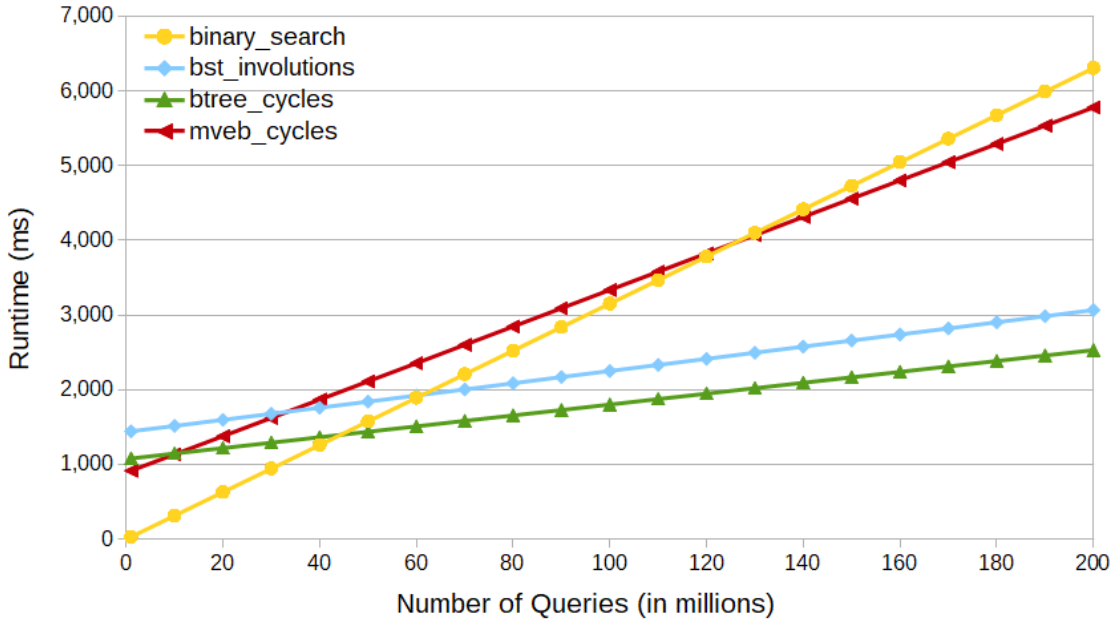

Figure 2.18: Combined time to permute and query each layout on the NVIDIA K40 with $N = 2^{29} - 1$ elements. The fastest permutation algorithm is used for each tree layout.

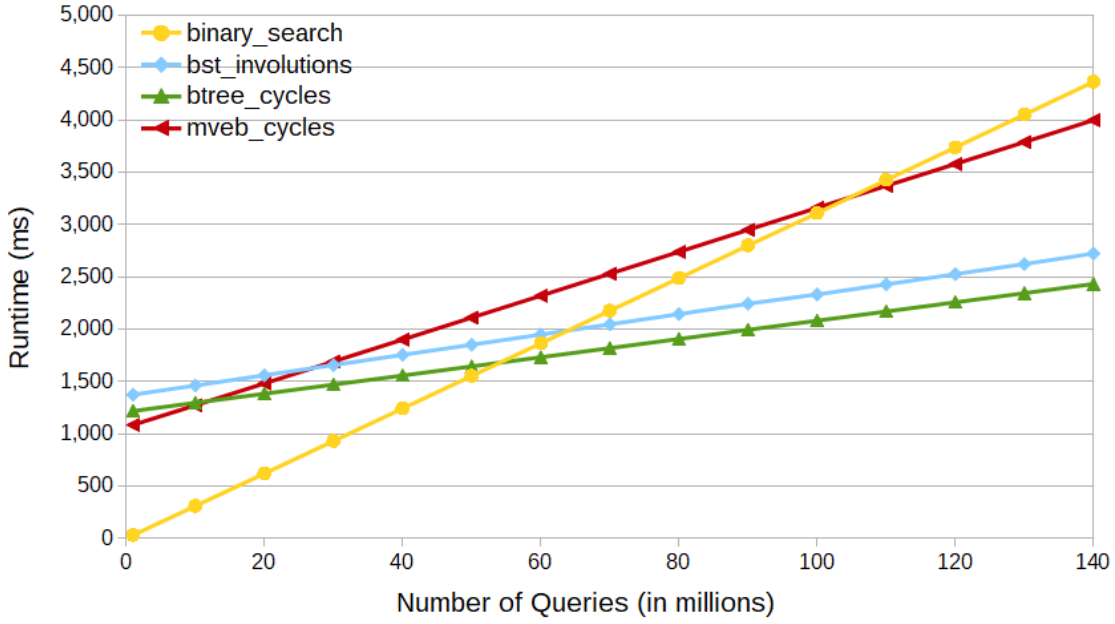

Figure 2.19: Combined time to permute and query each layout on the NVIDIA Quadro M4000 with $N = 2^{29} - 1$ elements. The fastest permutation algorithm is used for each tree layout.

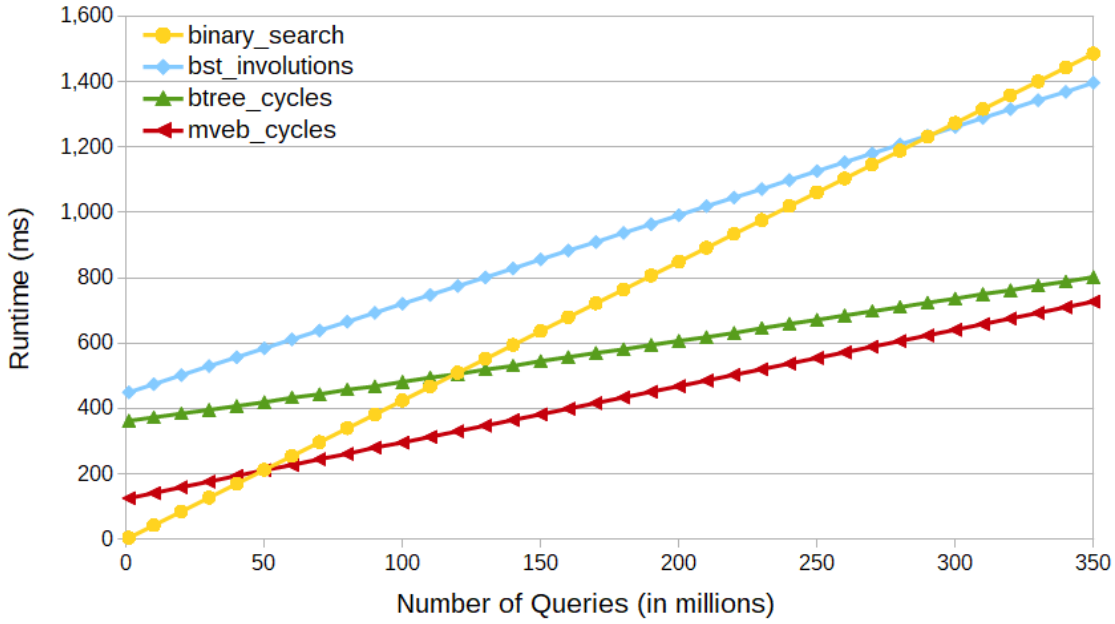

Figure 2.20: Combined time to permute and query each layout on the NVIDIA RTX 2080 Ti with $N = 2^{29} - 1$ elements. The fastest permutation algorithm is used for each tree layout.

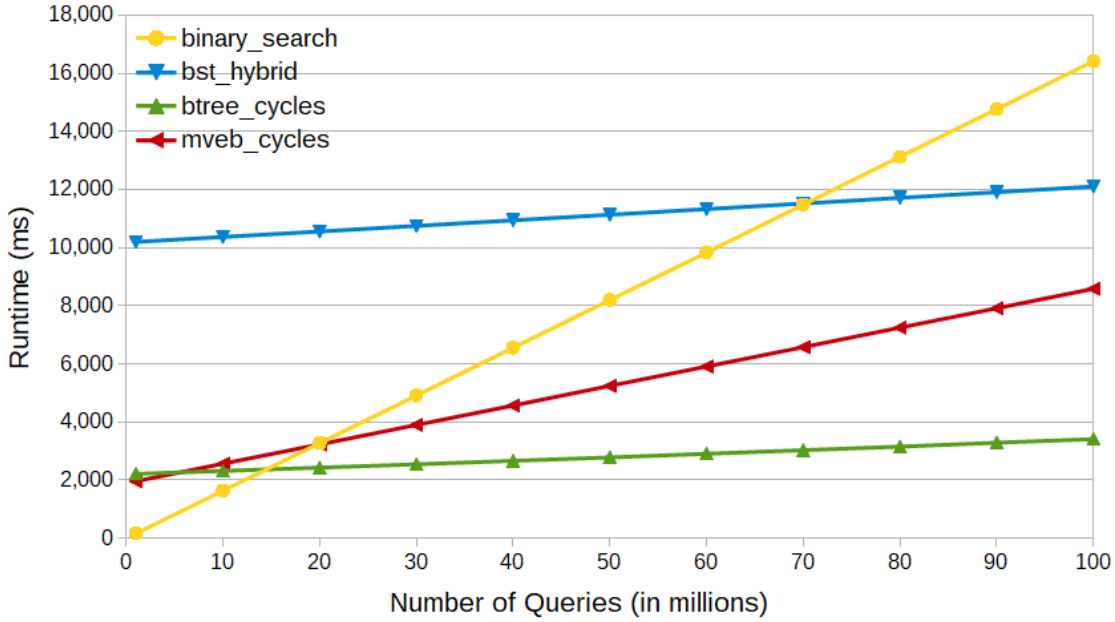

Figure 2.21: Combined time to permute and query each layout on the NVIDIA K40 with $N = 1$ billion elements. The fastest permutation algorithm is used for each tree layout.

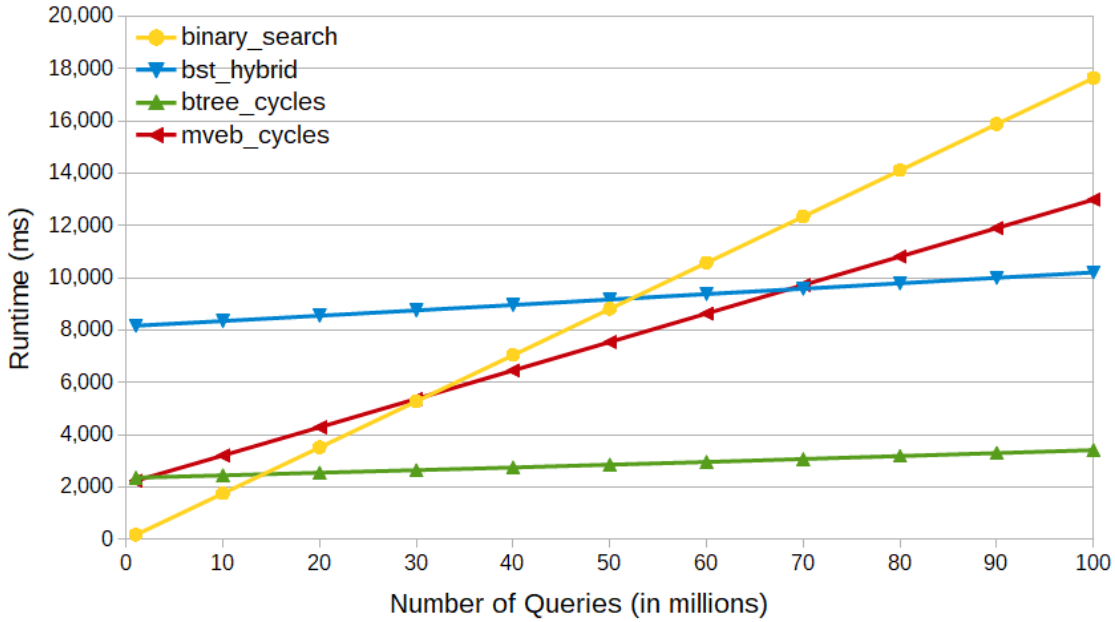

Figure 2.22: Combined time to permute and query each layout on the NVIDIA Quadro M4000 with $N = 1$ billion elements. The fastest permutation algorithm is used for each tree layout.

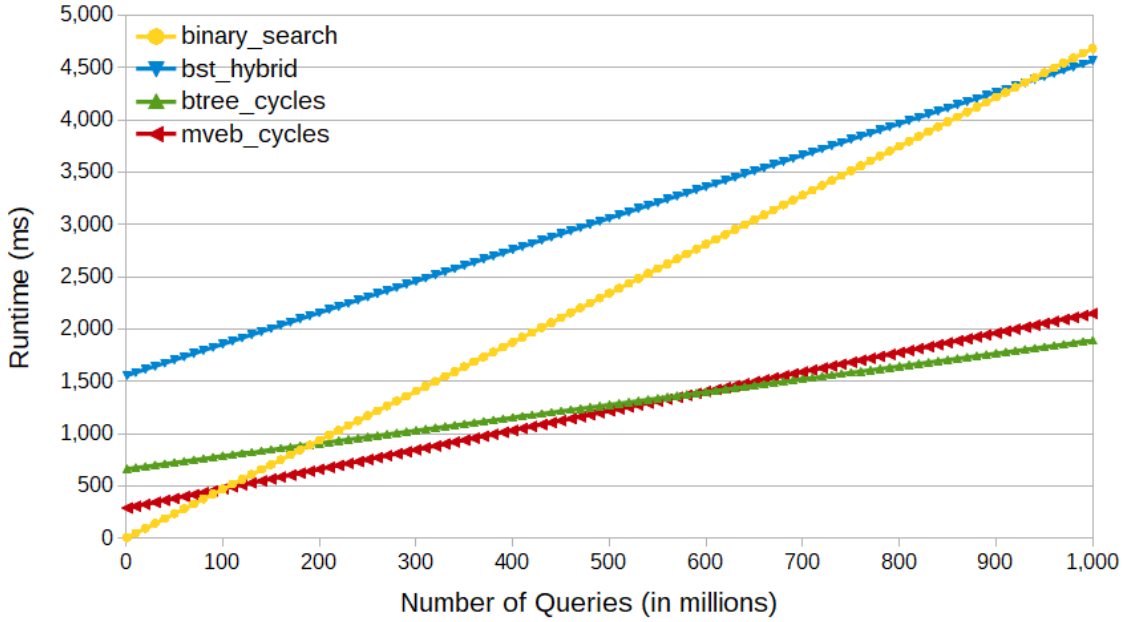

Figure 2.23: Combined time to permute and query each layout on the NVIDIA RTX 2080 Ti with $N = 1$ billion elements. The fastest permutation algorithm is used for each tree layout.

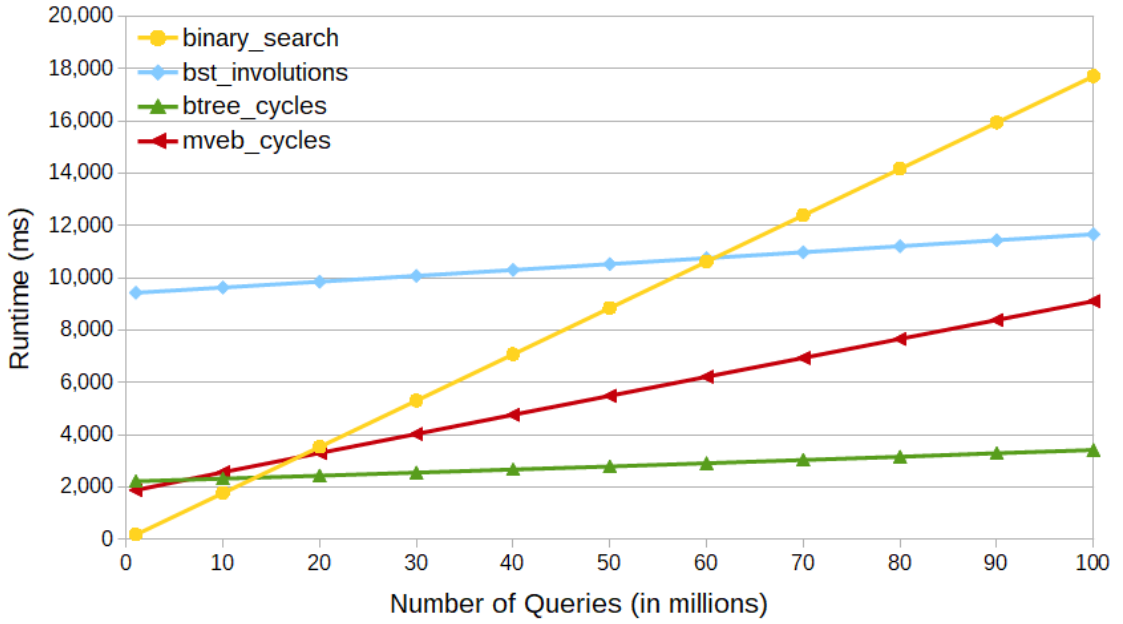

Figure 2.24: Combined time to permute and query each layout on the NVIDIA K40 with $N = 2^{30}-1$ elements. The fastest permutation algorithm is used for each tree layout.

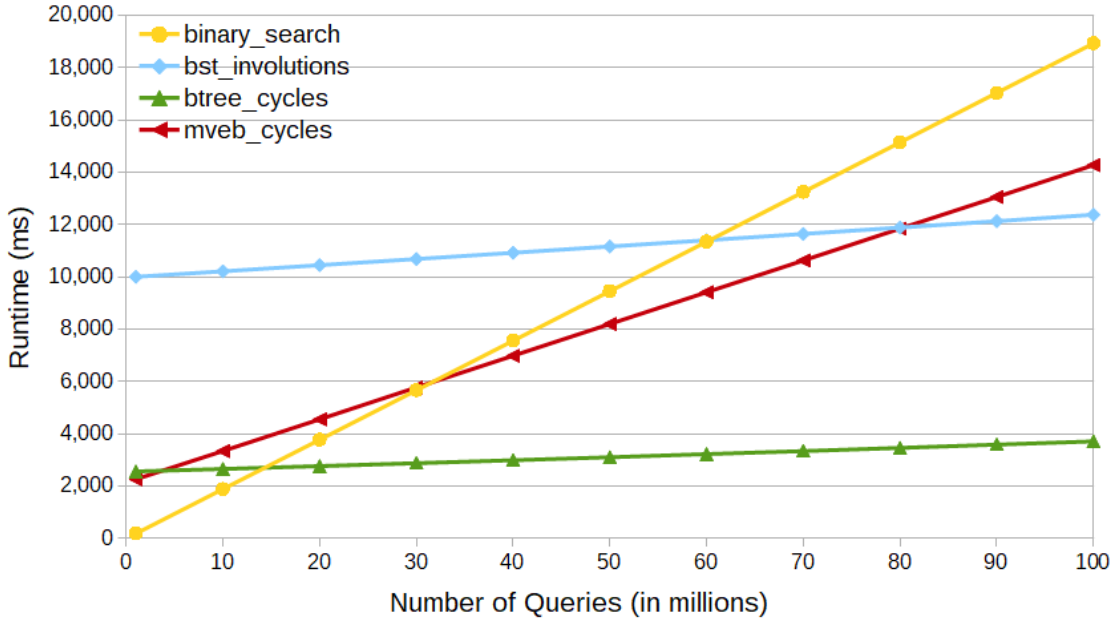

Figure 2.25: Combined time to permute and query each layout on the NVIDIA Quadro M4000 with $N = 2^{30} - 1$ elements. The fastest permutation algorithm is used for each tree layout.

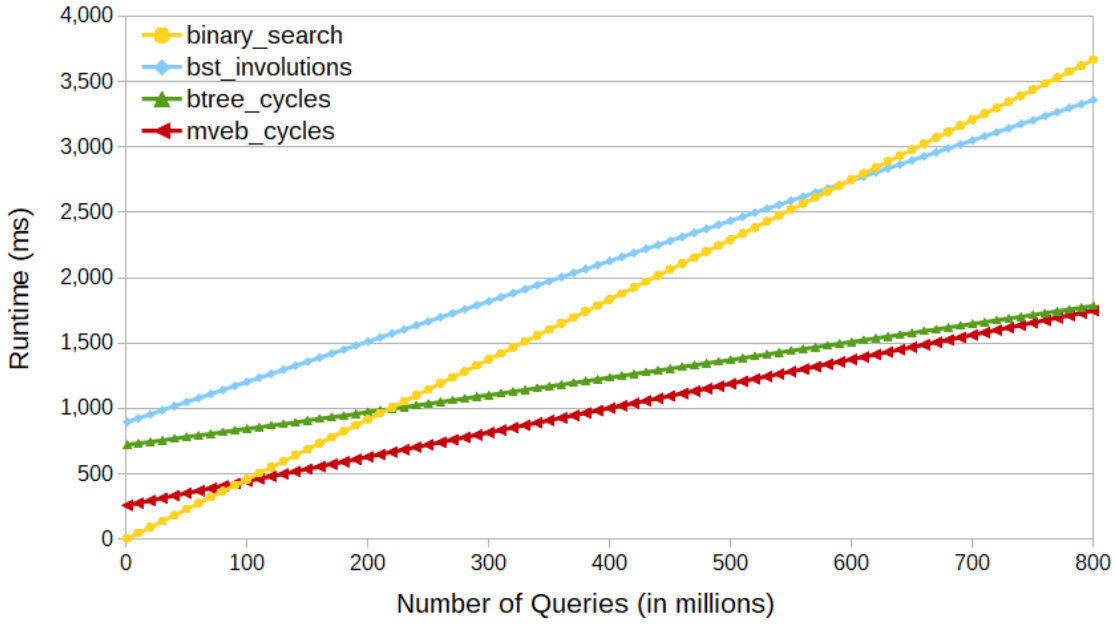

Figure 2.26: Combined time to permute and query each layout on the NVIDIA RTX 2080 Ti with $N = 2^{30} - 1$ elements. The fastest permutation algorithm is used for each tree layout.

CHAPTER 3
A PARALLEL PRIORITY QUEUE FOR SINGLE-SOURCE SHORTEST PATHS

Single-source shortest paths (SSSP) is a fundamental graph problem that has applications in many domains. Let $G = (V, E)$ be a directed graph consisting of $|V| = n$ vertices and $|E| = m$ non-negative weighted edges. Given a source vertex $v \in V$, the SSSP problem asks to find the minimum weight path from v to all other reachable vertices $u \in V$. In the sequential setting, the two classical solutions are Dijkstra's algorithm [35] and Bellman-Ford [9, 44]. Both take an iterative approach, where vertices are labeled with a tentative distance from the source vertex (initially set to $-\infty$) and are iteratively updated throughout execution. The difference in the algorithms comes in the order in which edges are processed. In each iteration of Dijkstra's algorithm, the outgoing edges of the minimum distance vertex, which has not been visited yet, are processed. Hence, Dijkstra's algorithm (using a Fibonacci heap) uses $O(m + n \log n)$ total operations. In contrast, Bellman-Ford performs $O(nm)$ total operations as all edges are processed in each iteration. Consequently, Bellman-Ford is easily parallelizable and is able to additionally compute shortest paths on graphs with negative edge weights.

In the parallel setting, a work-efficient algorithm with fast parallel runtime (e.g., $o(n)$) is yet to be developed. Instead, numerous parallel algorithms have been proposed that sacrifice work-efficiency for increased parallelism. Typically, these algorithms are designed to take advantage of specific properties of a particular class of graphs (e.g., random graphs, planar graphs, etc.).

The current state-of-the-art implementations of SSSP on GPUs are variations of the Delta-stepping algorithm of Meyer et al. [80], which has been proven to run well on graphs with random edge weights, "small" maximum vertex degrees, and "small" maximum shortest path lengths. In this work, we focus on graphs with sufficiently large diameter and degrees. We present a parallelization of the cache-oblivious bucket heap of Brodal et al. [15] and buffer heap of Chowdhury and Ramachandran [22] and adopt it for GPU architectures. Using the resulting heap, we implement a parallel variant of Dijkstra's algorithm and compare it to the state-of-the-art GPU SSSP implementations.

3.1 Preliminaries

3.1.1 Single-Source Shortest Paths

In the parallel setting, the Single-Source Shortest Paths (SSSP) problem suffers from the *transitive closure bottleneck* [64]. Thus, finding an algorithm that is work-efficient (i.e., the same work complexity as Dijkstra's algorithm) with $o(n)$ runtime on an arbitrary graph remains an important open problem. As a result, many alternative parallel SSSP algorithms have been proposed for

specific classes of graphs.

For planar graphs with integer edge weights between 0 and k, Klein and Subramanian [70] solve SSSP in $O(\text{polylog}\, n \log k)$ parallel time using n processors. Subramanian et al. [100] show that planar layered directed graphs can be decomposed using one-way separators that results in an SSSP algorithm with $O(\log^3 n)$ parallel runtime using n processors. Träff and Zaroliagis [102] use a region decomposition of a planar directed graph and show that for $0 < \epsilon < \frac{1}{2}$, their SSSP algorithm has $O((n^{2\epsilon} + n^{1-\epsilon}) \log n)$ depth and $O(n^{1+\epsilon})$ work. Atallah et al. [5] present a $O(\log^2 n)$ depth and $O(n \log n)$ work SSSP algorithm for planar layered directed graphs.

Chaudhuri and Zaroliagis [21] consider directed graphs with constant treewidth (a measure of how "close" the graph is to a tree) and use a $O(\log^2 n)$ depth and $O(n)$ work preprocessing stage that allows the computation of the SSSP with path length ℓ in $O(\alpha(n) \log n)$ depth and $O(\ell + \alpha(n) \log n)$ work[1].

For directed graphs with negative integer weights lower bounded by some integer $-k$, Cao et al. [18] present a parallelization of Goldberg's algorithm [52] that solves SSSP in $n^{5/4+o(1)} \log k$ depth and $\tilde{O}(m\sqrt{n} \log k)$ work, with high probability[2].

Crauser et al. [25] divide Dijkstra's algorithm into phases and show that on random graphs with random edge weights, the algorithm has $O(n^{\frac{1}{3}} \log n)$ depth and $O(m + n \log n)$ work with high probability[3] on a CRCW PRAM. Meyer et al. [80] introduce the Delta-stepping algorithm, where in each iteration, the outgoing edges of vertices within a distance interval of width Δ are processed. For an arbitrary graph with random edge weights, maximum degree d, maximum shortest path distance L, and $\Delta = \Theta\left(\frac{1}{d}\right)$; the Delta-stepping algorithm has a parallel depth of $O(dL \log n + \log^2 n)$ and total work of $O(n + m + dL \log n)$ on average.

For undirected graphs, Spencer and Shi [97] first compute the k nearest neighbors of every vertex in $O(\log n \log k)$ depth and $O(nk^2 \log n \log k + m)$ work and use this information to solve SSSP in $O(\frac{n}{k} \log n)$ depth and $O((m+nk) \log n)$ work. Blelloch et al. [12] combine the approaches of Spencer and Shi [97] and Meyer et al. [80] for undirected (k, ρ)-graphs, which are graphs where every vertex can reach its ρ closest neighbors in k or fewer edges traversed. Their SSSP algorithm has $O(\frac{kn}{\rho} \log n \log \rho L)$ depth and $O(km \log m)$ work. The authors additionally provide a preprocessing stage that transforms any undirected graph into a $(1, \rho)$-graph with at most $n\rho$ additional edges in $O(\rho \log \rho)$ depth and $O(m \log n + n\rho^2)$ work.

Considering approximate solutions on undirected graphs, Cohen [23] defines a (d, ϵ)-hop set of a graph, which augments the graph with new edges such that the shortest path, consisting of at most d edges, in the new graph has a distance within $(1 + \epsilon)$ of the shortest path in the original graph. Let $\epsilon_0 > 0$ be a fixed constant, Cohen presents a randomized algorithm that constructs a $(O(\text{polylog}\, n), O(1/\text{polylog}\, n))$-hop set with $O(n^{1+\epsilon_0})$ edges and uses it to solve the

[1] $\alpha(n)$ is the inverse Ackermann function

[2] $\tilde{O}$ hides polylogarithmic factors that may be present in the standard O notation

[3] For some constant $c > 0$, the probability is at least $1 - n^{-c}$

approximate shortest path problem from s different sources using $O(mn^{\epsilon_0} + s(m + n^{1+\epsilon_0}))$ work and polylogarithmic parallel runtime. Building on this work, Elkin and Neiman [37] devise an alternate randomized construction of a $(O(1), O(1/\text{polylog } n))$-hop set with $O(n^{1+\epsilon_0} \log n)$ edges that is used to improve the parallel runtime of the approximate shortest path problem. And in 2019, Elkin and Neiman [38] present a randomized construction of a $(O(1), O(1/\text{polylog } n))$-hop set with $O(n^{1+\epsilon_0} \log^* n)$ edges.

Within the context of GPUs, Harish et al. [57] showed that a GPU implementation of Bellman-Ford outperforms a sequential CPU approach. More recently in 2014, Davidson et al. [31] experimentally evaluated several GPU implementations of SSSP. Notably, a variation of Bellman-Ford, called Workfront Sweep, and an implementation of the Delta-stepping algorithm of Meyer et al. [80], called Near-Far. Workfront Sweep uses a heuristic that seeks to reduce the amount of work performed in each iteration of Bellman-Ford by only processing edges outgoing from vertices whose tentative distances were updated in the previous iteration. Let W be the average weight of edges in the graph, d' be the average degree in the graph, and w be the number of threads in a warp of a GPU. The Near-Far implementation uses $\Delta = \frac{W \cdot w}{d'}$ and only two buckets (the "near" and "far" buckets). Experiments were conducted on 8 different graphs and results showed that the Near-Far implementation provides performance gains on 6 graphs with low diameter and degree (5 out of the 6 graphs have random edge weights). Building on the Workfont Sweep approach, Busato and Bombieri [17] provide a variation of Bellman-Ford that additionally classifies edges based on the operations needed to process the edge (e.g., whether an atomic operation is needed). In 2016, Wang et al. [107] introduced the Gunrock library that contains an implementation of the Near-Far approach and showed that on graphs with low degree and random edge weights between 1 and 64, their implementation outperforms both CPU and GPU libraries for SSSP. Lastly in 2021, Wang et al. [105] improved on the Near-Far approach by using a heuristic that periodically changes the Δ value, adding a dynamic memory allocator to allow for the use of multiple buckets, and using a designated group of threads to manage the coordination of work.

3.1.2 Priority Queue

Dijkstra's algorithm for solving SSSP relies on an efficient priority queue to find the vertex with the minimum tentative distance that has not been visited yet. Formally, the priority queue ADT is defined over a collection of elements, Q, where each element e consists of a value and priority, i.e., $e \in Q = (val, p)$. For each element e, we define $e.val$ and $e.p$ to be the value and priority, respectively. The ADT supports the following operations:

- EXTRACTMIN removes and returns the element in Q with the smallest priority, i.e., $e \in Q$ such that $e.p = \min_{e_i \in Q} e_i.p$
- UPDATE(e) adds new element $e = (val, p)$ to Q, and if there exists an $e' = (val, p')$ with the

same value in Q, then remove e' (so that it is replaced by e)[4]

- DELETE(e) removes e from Q if it is contained in Q

There are many data structures defined that implement the priority queue ADT, including various types of heaps [15, 16, 22, 24, 34, 47, 48]. We note that many other data structures, including binary search trees or sorted arrays, can be used as priority queues, though they provide additional functionality and are therefore not as efficient when performing only the above operations. Though not considered part of the standard priority queue ADT, in this work we additionally consider the BULKUPDATE(U) operation that, given a set of elements U, performs UPDATE(e) for each $e \in U$. In each iteration of Dijkstra's algorithm, all outgoing edges of the current minimum distance vertex are processed, resulting in a set of new (shorter) tentative distances. The BULKUPDATE(U) operation is used to update the tentative distances of these vertices in a single efficient operation, thereby allowing efficient processing of graphs with large degrees.

Several fundamental priority queue data structures provide tradeoffs between simplicity and performance (e.g., binary heaps, Fibonacci heaps [48], pairing heaps [47], etc.). However, these heaps are inherently sequential and operations on heaps cannot be easily parallelized. Thus, several priority queues have been developed to expose parallelism [16, 34, 59, 60, 95]. Brodal et al. [16] presents a structure that performs all standard priority queue operations in constant time and logarithmic work in the PRAM model. Hübschle-Schneider et al. [59] design a randomized parallel priority queue that supports BULKUPDATE on up to d elements in $O(1 + \log d)$ parallel time, in a parallel distributed memory model (i.e., processors communicate over an interconnection network).

To our knowledge, all existing parallel priority queue data structures are not cache-efficient, and as such may not perform well on GPUs or other parallel systems that rely on locality of reference to achieve peak performance. While not inherently parallel, in the context of the External Memory or cache-oblivious models, the cache-oblivious bucket heap [15] and buffer heap [22] structures achieve sub-constant amortized time operations when the block size, B, is sufficiently large (see Section 1.3.2 for a definition). Since there are no parallel, cache-efficient priority queue structures, few works have considered using priority queues on GPUs: He et al. [58] present a priority queue that achieves a speed up factor of 30 over sequential execution; and Baudis et al. [6] demonstrate that for small queues of up to 500 items, simple circular buffers outperform tree-based queues when evaluated on discrete event simulation and A* search.

Data structure	Time		I/O Complexity		Total I/Os
	ExtractMin	BulkUpdate	ExtractMin	BulkUpdate	$\frac{n}{d}$ BulkUpdates
Seq. Bucket Heap [15]	$O(\log n)$	$O(d \log n)$	$O\left(\frac{1}{B} \log \frac{n}{B}\right)$	$O\left(\frac{d}{B} \log \frac{n}{B}\right)$	$O\left(\frac{n}{B} \log \frac{n}{B}\right)$
Parallel Prio. Queue [16]	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(n \log \frac{n}{d})$
Bulk Parallel Prio. Queue [59]	$O(1 + \log d)$	$O(1 + \log d)$	-	-	-
This work	$O(1 + \log d)$	$O(1 + \log d)$	$O\left(\frac{\log (n/d)}{pB} + \frac{1}{B}\right)$	$O\left(\frac{d \log (n/d)}{pB} + \frac{d}{B}\right)$	$O\left(\frac{n}{B} \log \frac{n}{d}\right)$

Table 3.1: Comparison of priority queue operations in different sequential and parallel models: n is the number of input elements, $d \leq n$ is then maximum number of elements supported by BulkUpdate, p is the number of processors, and B is the width of data transfers to external memory. The right-most column shows the total number of I/Os when performing $\frac{n}{d}$ BulkUpdate operations, each consisting of d updates. (We note that the Bulk Parallel Priority Queue [59] is designed in a parallel distributed memory model, hence, does not include I/Os to external memory.)

3.2 Contributions

We present the parallel bucket heap, denoted parBucketHeap, an I/O efficient parallel priority queue designed for GPU architectures supporting the BulkUpdate operation. Using the parBucketHeap, the number of I/Os performed for a sequence of BulkUpdate operations is significantly reduced compared to the current best data structures (see Table 3.2). The BulkUpdate operation is particularly useful when the parBucketHeap is used to solve the SSSP problem using Dijkstra's algorithm, as batches of update operations are performed when the vertex being processed has multiple outgoing edges.

We use the parBucketHeap to implement a parallel version of Dijkstra's algorithm, denoted parDijsktra. Both parBucketHeap and parDijsktra are implemented using CUDA C/C++ [87]. Experiments are conducted on 2 NVIDIA GPUs: an RTX 2080 Ti and a Quadro M4000. We compare the performance of parDijsktra to the current state-of-the-art SSSP GPU implementations: Gunrock [107] and Asynchronous Dynamic Delta-Stepping (ADDS) [105]. Our results show that for sufficiently dense graphs with large diameter ($n = 30,000$ vertices and diameter $n - 1$), parDijsktra using the parBucketHeap has a peak speed up of 2.8 and 12 over Gunrock and ADDS, respectively, on the RTX 2080 Ti; and a peak speed up of 5.4 over Gunrock on the Quadro M4000 (ADDS does not support the Quadro M4000).

The paper is organized as follows: in Section 3.3 we provide an overview of the sequential bucket heap and present our parBucketHeap data structure; in Section 3.4 we analyze the parBucketHeap in the CREW PRAM and PEM models; in Section 3.5 we provide implementation details and experimental results; and lastly, in Section 3.6 we conclude with a brief summary.

[4] In this work, we assume that updates only decrease priority. However, this assumption can be removed by adding a timestamp to each element when it is inserted into the structure; and when deleting duplicate entries, the most recent timestamp is kept.

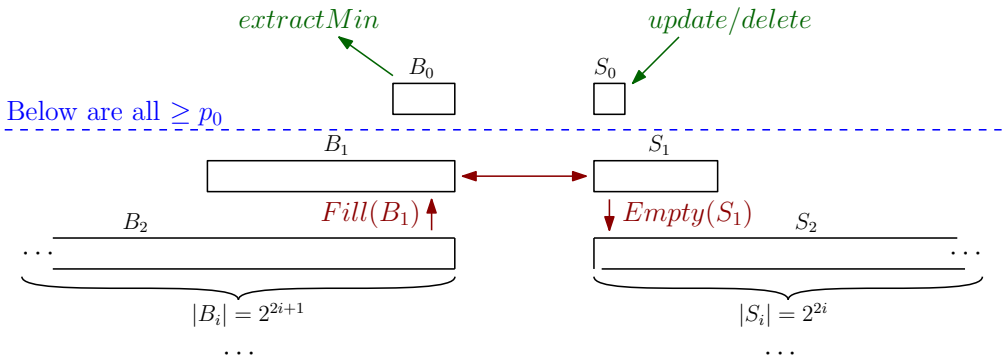

Figure 3.1: Illustration of the sequential bucket heap structure of Brodal et al. [15]. UPDATEs and DELETEs are inserted into S_0, while EXTRACTMINs are removed from B_0. EMPTY(S_i) empties S_i into S_{i+1}, FILL(B_i) fills B_i from B_{i+1}, while p_i is maintained at each level to ensure the heap property between levels.

3.3 Bucket Heap

The parallel bucket heap is a parallelization of the cache-oblivious bucket heap of Brodal *et al.* [15] and buffer heap of Chowdhury and Ramachandran [22]. In this work, we follow the naming conventions and presentation of the bucket heap, thus, we first provide a general overview of the sequential bucket heap.

3.3.1 Sequential Bucket Heap

The bucket heap is a hierarchical data structure, where each level consists of a *bucket* and a *signal buffer*. We note that elements are always stored in sorted order by value (not priority) in each bucket and signal buffer. Figure 3.1 illustrates the sequential bucket heap structure and shows the relationship between elements stored at each level. Elements inserted into the bucket heap (via UPDATE operations) are moved into the top level's signal buffer; and elements removed from the bucket heap (via EXTRACTMIN operations) are taken from the top level's bucket. For any given level, if its signal buffer becomes sufficiently full (e.g., at least half full), then it is emptied into the bucket on the same level and overflow elements are merged into the next (lower) level's signal buffer. And if the bucket becomes too empty (e.g., at least half empty), then it is filled with the smallest priority elements from the next (lower) level's bucket and signal buffer.

Formally, a bucket heap storing n elements has $q = \lceil \log_4 n \rceil + 1$ levels, where for each level $i \in \{0, 1, \ldots, q-1\}$, the maximum capacity of the i-th level's bucket, denoted B_i, and signal buffer, denoted S_i, is 2^{2i+1} and 2^{2i}, respectively. The bucket heap maintains the invariant that for all $j > i$, all elements in B_i have a smaller (or equal) priority than all elements in B_j. This ensures that if B_0 is non-empty and S_0 is empty, then B_0 will contain the minimum priority element in the structure. Therefore, if this condition is satisfied, EXTRACTMIN can simply remove and return the minimum

priority element in B_0. Furthermore, as long as S_0 is kept non-full, UPDATE(e) can simply insert e into S_0. DELETE operates similar to UPDATE using an element with a special priority value, DEL, that moves down the structure, removing elements with matching key values.

The bucket heap transfers elements between levels via the EMPTY and FILL operations. The EMPTY(S_i) operates as follows: (1) scan B_i to find the element with maximum priority, denoted p_i; (2) merge elements in S_i with B_i; (3) for any elements with duplicate values, remove those with larger priority; and (4) all elements e such that $e.p > p_i$ are merged into S_{i+1}. If the resulting number of elements in B_i is too full (i.e., there are too many elements with $e.p \leq p_i$) then p_i is updated so that $|B_i| = 2^{2i+1}$ and the elements with priorities larger than p_i are merged into S_{i+1}. Updating the priority of an existing element is accomplished when elements with duplicate values are found and the element with larger priority is removed (elements with special delete priorities, DEL, are also applied this way). Since lists are stored sorted by value, elements with duplicate values are stored next to each other and can be removed with a scan. If the number of elements in a bucket B_i fall under the minimum size (e.g., half full), FILL(B_i) is called, which empties S_i into B_i and fills any remaining space in B_i with elements from level $(i+1)$. This is accomplished by: (1) calling EMPTY(S_i); (2) if B_i is non-full and S_{i+1} is non-empty, then EMPTY(S_{i+1}) is called; and (3) if B_i is non-full, then B_i is filled with the smallest priority elements in B_{i+1}. All of these operations are performed via scans of contiguous arrays, leading to $O\left(\frac{1}{B}\log\frac{n}{B}\right)$ amortized I/O complexity of the EXTRACTMIN, UPDATE, and DELETE operations of the sequential bucket heap.

3.3.2 Parallel Bucket Heap

We parallelize the sequential bucket heap by using parallel variants of EMPTY and FILL; and allowing non-adjacent levels to execute in parallel. Additionally, we increase the maximum capacity of every bucket and signal buffer by a factor of d, hence, $|B_i| = d \cdot 2^{2i+1}$ and $|S_i| = d \cdot 2^{2i}$. When U is sorted by value, a BULKUPDATE(U) of up to d elements can be efficiently performed by simply inserting all updates into S_0. By increasing the capacity of all buckets and signal buffers, we decrease the total number of levels of the bucket heap to $q = \left\lceil \log_4 \frac{n}{d} \right\rceil + 1$.

For ease of exposition, we combine the EMPTY and FILL operations into a single operation, RESOLVE (Algorithm 1). Let ℓ be the maximum non-empty level of the parallel bucket heap. The RESOLVE(i) operation empties S_i and fills B_i, leaving S_i empty and B_i full (unless $i = \ell$). Our description of the RESOLVE operation in Algorithm 1 is high-level and the subroutines MERGE, DELETEDUPLICATES, and SELECT can be implemented in different ways, depending on the desired level of parallelism, which we discuss in our analysis in the subsequent sections.

Parallel Execution Sequence

Consider a series of N *operations*, defined as $\text{OP}_1, \text{OP}_2, \ldots, \text{OP}_N$, where each operation is EXTRACTMIN, UPDATE, DELETE, or BULKUPDATE. In the sequential setting, the bucket heap empties signal

Algorithm 1 RESOLVE(i)

Precondition: if $i < \ell$, then $|B_i| \geq d \cdot 2^{2i}$
Precondition: $|S_i| \leq d \cdot 2^{2i}$
Precondition: if $i+1 < \ell$, then $|B_{i+1}| \geq d \cdot 2^{2(i+1)} + d \cdot 2^{2i}$
Precondition: $|S_{i+1}| \leq d \cdot 2^{2(i+1)} - d \cdot 2^{2i}$
Postcondition: if $i < \ell$, $|B_i| = d \cdot 2^{2i+1}$
Postcondition: $|S_i| = 0$

1: **if** $|S_i| > 0$ ▷ Empty S_i if needed
2: $B_i \leftarrow \text{MERGE}(S_i,\, B_i)$; $S_i = \emptyset$
3: $B_i \leftarrow \text{DELETEDUPLICATES}(B_i)$
4: $num \leftarrow |\{e : e \in B_i \text{ and } e.p \leq p_i\}|$ ▷ Count elements with small priority
5: **if** $num > 2^{2i+1}$ ▷ Update p_i if needed
6: $p_i \leftarrow \text{SELECT}(B_i, 2^{2i+1})$
7: $B'_i = \{e : e \in B_i \text{ and } e.p > p_i\}$ ▷ Move large priority elements to S_{i+1}
8: $B_i = \{e : e \in B_i \text{ and } e.p \leq p_i\}$
9: $S_{i+1} \leftarrow \text{MERGE}(S_{i+1}, B'_i)$

10: **if** $|B_i| < 2^{2i+1}$ and i is not largest non-empty level ▷ Fill B_i if needed
11: $B_{i+1} \leftarrow \text{MERGE}(B_{i+1}, S_{i+1})$; $S_{i+1} \leftarrow \emptyset$
12: $B_{i+1} \leftarrow \text{DELETEDUPLICATES}(B_{i+1})$
13: $p_i \leftarrow \text{SELECT}(B_{i+1}, 2^{2i+1} - |B_i|)$
14: $B'_{i+1} \leftarrow \{e : e \in B_{i+1} \text{ and } e.p \leq p_i\}$ ▷ Pull elements up to fill B_i
15: $B_{i+1} \leftarrow \{e : e \in B_{i+1} \text{ and } e.p > p_i\}$
16: $B_i \leftarrow \text{MERGE}(B_i, B'_{i+1})$
17: $S_{i+1} \leftarrow \{e : e \in B_{i+1} \text{ and } e.p > p_{i+1}\}$ ▷ Move large priority elements back to S_{i+1}

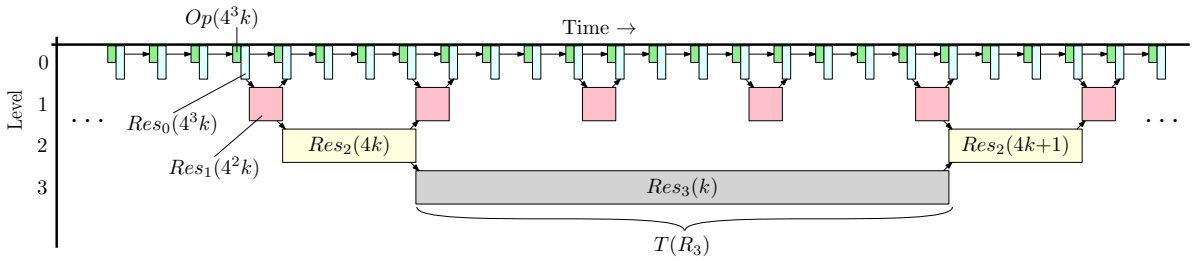

Figure 3.2: Illustration of the dependencies when performing a series of operations. Small green boxes represent operations (extractMin, update, delete, or bulkUpdate) and the remaining boxes represent Resolves. Dependencies are shown with arrows between boxes. Since resolving larger levels takes more time, we represent them by wider boxes which are scaled to show that, if Resolve(0) takes d time then Resolve(i) can take $d \cdot 2^{2i}$ time without delaying any operations (which occur every $5d$ parallel memory accesses).

buffers and fills buckets as needed. While in the parallel setting, we can proactively perform Resolve on different levels of the bucket heap in parallel. Let $Res_i(k)$ be the k-th execution of Resolve(i) during the series of operations.

We define $A \to B$ to denote that task B depends on task A being completed in order for the preconditions of task B to be satisfied. Intuitively, we view the execution of Resolves as a directed acyclic graph (DAG) where each vertex represents a Resolve operation and each edge is a dependency. Figure 3.2 illustrates this DAG, where green boxes represent operations, other color boxes represent Resolves, and the width of each box is the amount of time needed to perform it.

Theorem 20. *Let $i > 0$ and $k \geq 1$. If, for every level i, we perform* Resolve(i) *after every fourth* Resolve($i-1$)*, then all preconditions are always satisfied, i.e.,*

$$Res_{i-1}(4k) \to Res_i(k)$$

Proof. After Resolve(i) completes, $|S_i| = 0$ and $|B_i| = d \cdot 2^{2i+1}$. Each call to Resolve($i-1$) adds at most $d \cdot 2^{2(i-1)}$ elements to S_i and removes at most $d \cdot 2^{2(i-1)}$ elements from B_i. Hence, after 4 executions of Resolve($i-1$), $|S_i| \leq 4d \cdot 2^{2(i-1)} = d \cdot 2^{2i}$ and $|B_i| \geq d \cdot 2^{2i+1} - 4d \cdot 2^{2(i-1)} = d \cdot 2^{2i}$. A similar argument is made for the preconditions on B_{i+1} and S_{i+1}. □

After each operation, we must perform a Resolve(0) to ensure that the preconditions (S_0 is empty and B_0 is non-empty) are met for the next operation. Thus, each Op_k depends on $Res_0(k-1)$, i.e., $Res_0(k-1) \to \text{Op}_k \to Res_0(k) \Rightarrow Res_0(k-1) \to Res_0(k)$. Furthermore, recall that Resolve(i) can modify B_i, S_i, B_{i+1}, and S_{i+1}, therefore, concurrent access to these arrays need to be avoided during parallel executions of Resolve. In other words, no two consecutive levels can execute Resolve concurrently, i.e., $Res_i(k) \to Res_{i-1}(4k+1)$.

3.4 Analysis

3.4.1 PRAM Analysis

Lemma 21. *Let $D(R_i)$ be the depth and $W(R_i)$ be the work of* Resolve(i)*. For all $i \geq 0$ and $d \geq 1$, $D(R_i) = O(\max(1, i + \log d))$ and $W(R_i) = O\left(d \cdot 2^{2i}\right)$.*

Proof. The Resolve operation relies on performing Merge, Select, and DeleteDuplicates on elements in levels i and $i+1$. DeleteDuplicates involves identifying and deleting duplicate entries and compressing the remaining elements into contiguous space, which can be accomplished via a parallel scan and prefix sum. Hence, Merge, Select, and DeleteDuplicates on n total elements can be performed with $D(n) = O(\log n)$ depth and $W(n) = O(n)$ work [62]. Therefore, $D(R_i) = O(\log(|B_{i+1}| + |S_{i+1}|)) = O(i + \log d)$ and $W(R_i) = O(|B_{i+1}| + |S_{i+1}|) = O\left(d \cdot 2^{2i}\right)$. □

Lemma 22. *Let $c > 0$ be some constant, $T(R_0)$ be the time it takes to execute* Op(k) *and* Resolve(0)*, and $T(R_i)$ be the time it takes to execute* Resolve(i)*. For any $T(R_i) \leq cd \cdot 2^{2i}$, $Res_i(k)$ completes execution before time*

$$\left(5k \cdot 4^i - 5 + \frac{1}{3}(4^i - 1)\right) \cdot T(R_0) + T(R_i)$$

Proof. We know from the dependencies that for $k \pmod 4 \not\equiv 1$, $Res_i(k)$ cannot start until $Res_{i-1}(4k)$ finishes execution; and for $k \pmod 4 \equiv 1$, $Res_i(k)$ cannot start until $Res_{i-1}(4k)$ and $Res_{i+1}((k-1)/4)$ finishes execution. Using induction, $Res_{i-1}(4k)$ completes execution before time

$$\begin{aligned}
&\left(5(4k) \cdot 4^{i-1} - 5 + \frac{1}{3}(4^{i-1} - 1)\right) \cdot T(R_0) + T(R_{i-1}) \\
&\leq \left(5k \cdot 4^i - 5 + \frac{1}{3}(4^{i-1} - 1)\right) \cdot cd + \left(cd \cdot 2^{2(i-1)}\right) \\
&= \left(5k \cdot 4^i - 5 + \frac{1}{3}(4^{i-1} - 1) + 2^{2(i-1)}\right) \cdot cd \\
&= \left(5k \cdot 4^i - 5 + \frac{1}{3}(4^i - 1)\right) \cdot T(R_0)
\end{aligned}$$

and $Res_{i+1}((k-1)/4)$ completes execution before time

$$\begin{aligned}
&\left(5\left((k-1)/4\right) \cdot 4^{i+1} - 5 + \frac{1}{3}(4^{i+1} - 1)\right) \cdot T(R_0) + T(R_{i+1}) \\
&\leq \left(5k \cdot 4^i - 5 \cdot 4^i - 5 + \frac{1}{3}(4^{i+1} - 1)\right) \cdot cd + \left(cd \cdot 2^{2(i+1)}\right) \\
&= \left(5k \cdot 4^i - 5 \cdot 4^i - 5 + \frac{1}{3}(4^{i+1} - 1) + 2^{2(i+1)}\right) \cdot cd
\end{aligned}$$

$$= \left(5k \cdot 4^i - 5 + \frac{1}{3}(4^i - 1)\right) \cdot T(R_0)$$

Therefore, $Res_i(k)$ completes execution before time $\left(5k \cdot 4^i - 5 + \frac{1}{3}(4^i - 1)\right) \cdot T(R_0) + T(R_i)$. □

Theorem 23. EXTRACTMIN, UPDATE, DELETE, *and* BULKUPDATE *on up to d elements, has an amortized parallel depth of $O(1 + \log d)$ and $O\left(d \log_4 \frac{n}{d}\right)$ work.*

Proof. From Lemma 22, N operations complete execution before time $(5N - 4) \cdot T(R_0)$. On a machine with infinite processors, $T(R_0) = D(R_0) = O(1 + \log d)$, and the depth is $O(N \log d)$ or an amortized $O(1 + \log d)$ per operation. Since the PARBUCKETHEAP has a total of $\lceil \log_4 \frac{n}{d} \rceil + 1$ levels, $O(\log_4 \frac{n}{d})$ levels may be active in each step. Hence, for a single processor, $T(R_0) = W(R_0) = O(d)$, and the PARBUCKETHEAP performs a total of $O\left(Nd \log_4 \frac{n}{d}\right)$ work or an amortized $O\left(d \log_4 \frac{n}{d}\right)$ per operation. □

3.4.2 I/O Analysis

To optimize the I/O performance of the PARBUCKETHEAP, we set $d = O(M)$, so that S_0, B_0, and an additional buffer of size d can always be maintained in a single processor's internal memory space. For a single processor, RESOLVE(i) can be performed using scans of contiguous memory, hence, RESOLVE(i) performs $O\left(\frac{d \cdot 2^{2i}}{B}\right)$ I/Os.

Theorem 24. *For $1 \leq p \leq \lceil \log_4 \frac{n}{d} \rceil + 1$, the* PARBUCKETHEAP *can perform* EXTRACTMIN, DELETE *and* UPDATE *using*

$$O\left(\frac{\log_4 n/d}{pB} + \frac{1}{B}\right)$$

amortized parallel I/Os.

Proof. Let $p = \lceil \log_4 \frac{n}{d} \rceil + 1$. We assign processor p_i to level i of the PARBUCKETHEAP. In particular, processor p_0 is assigned to the first level of the heap and it always maintains S_0, B_0, and the auxiliary buffer of size d in internal memory. This additional buffer of size d is used as an intermediate storage space for elements that will be merged into S_1 during the next call to RESOLVE(0). Hence, the first level of the heap is able to process $\Theta(d)$ EXTRACTMIN, DELETE and UPDATE operations in internal memory before calling RESOLVE(0). We apply the resolution schedule described in Section 3.4.1, where $T(R_0) \leq \frac{cd}{B}$ is the number of parallel I/Os performed by RESOLVE(0). As we perform $\Theta(d)$ operations for each RESOLVE(0), N operations can be performed using N/d calls to RESOLVE(0). Thus, $Res_0(N/d)$ finishes execution before time $\left(\frac{5N}{d} - 4\right) \cdot T(R_0)$. Therefore, performing N operations takes $O\left(\frac{N}{d} \cdot \frac{d}{B}\right) = O\left(\frac{N}{B}\right)$ parallel I/Os; or $O\left(\frac{1}{B}\right)$ per operation.

Let $1 \leq p < \lceil \log_4 \frac{n}{d} \rceil + 1$. Similar to the first case, we assign processor p_0 to the first level of the heap. The remaining levels are divided equally across the remaining processors, so that each processor (except p_0) maintains $O\left(\frac{\log_4 n/d}{p}\right)$ levels. In the resolution schedule, each processor

performs all of the work associated with the levels it is assigned. Therefore, performing N operations takes $O\left(\frac{N}{d} \cdot \frac{d}{B} \cdot \frac{\log_4 n/d}{p}\right) = O\left(\frac{N}{pB} \cdot \log_4 \frac{n}{d}\right)$ parallel I/Os; or $O\left(\frac{\log_4 n/d}{pB}\right)$ per operation. □

Theorem 25. *For $1 \leq p \leq \lceil \log_4 \frac{n}{d} \rceil + 1$, the* parBucketHeap *can perform* BulkUpdate(U) *on up to d elements using*

$$O\left(\frac{d}{pB} \cdot \log_4 n/d + \frac{d}{B}\right)$$

amortized parallel I/Os.

Proof. From Theorem 24, it follows that N BulkUpdate(U) and Resolve(0) operations finishes execution before time $(5N-4) \cdot T(R_0)$. Therefore, for $p = \lceil \log_4 \frac{n}{d} \rceil + 1$, performing N operations takes $O\left(\frac{Nd}{B}\right)$ parallel I/Os; or $O\left(\frac{d}{B}\right)$ per operation. And for $1 \leq p < \lceil \log_4 \frac{n}{d} \rceil + 1$, performing N operations takes $O\left(\frac{Nd}{B} \cdot \frac{\log_4 n/d}{p}\right) = O\left(\frac{Nd}{pB} \cdot \log_4 n/d\right)$ parallel I/Os; or $O\left(\frac{d}{pB} \cdot \log_4 n/d\right)$ per operation. □

3.5 Experiments

3.5.1 Implementation Details

From the analysis performed in Section 3.4.1, the parBucketHeap is able to support a large number of threads in the PRAM model. However, the parallel I/O performance (in Section 3.4.2) relies on a relatively small number of processors, $1 \leq p \leq \lceil \log_4 \frac{n}{d} \rceil + 1$ (i.e., at most 1 processor per level of the parBucketHeap), due to a single processor needing to maintain the first level of the heap in internal memory. This restriction of processors in the PEM model can be an issue in traditional many-core architectures (e.g., multi-core CPU systems), however, the thread and memory hierarchy of GPUs allows us to harness both the parallelism shown in the PRAM analysis and I/O efficiency shown in the PEM analysis. As all I/Os performed in the parBucketHeap are scans of contiguous memory locations, we are able to schedule warps such that warps belonging to the same thread block access contiguous blocks of w elements. Hence, we map each thread block consisting of tw threads (or t warps) to a single PEM processor (where shared memory is used as the internal memory space) and use a block size of $B = tw$.

At the start of execution, all thread blocks are launched onto the GPU and execute Empty(S_i) and Fill(B_i) as needed throughout the course of the program. Because CUDA C/C++ only provides hardware synchronization primitives between threads within a thread block (i.e. intra-block synchronization), synchronization between threads across thread blocks (i.e., inter-block synchronization) can only be performed via software implemented synchronizations. Past implementations of software synchronizations between thread blocks [50, 108, 109] show that variations of busy-wait (i.e., spin locks) can be used to communicate synchronization information (e.g. the number of available resources or the number of thread blocks that have reached the synchronization point).

We use a similar approach, where each thread block (i.e., level of the parBucketHeap) has a designated global memory location that is used to signal the particular thread block to execute Empty(S_i) and/or Fill(B_i). Hence, a single thread per thread block can be used to continuously check this memory location until it has been set to a particular value. To avoid deadlocks, this approach requires that all thread blocks are able to be concurrently scheduled onto the GPU. A simple way to ensure this is to never launch more thread blocks than there are SMs.

As shown in the psuedocode of Resolve(i) (Algorithm 1), implementing Empty(S_i) and Fill(B_i) requires using parallel subroutines: Merge, PrefixSums, and Select. We use the implementation of Merge provided in the Thrust library [88] and the implementation of PrefixSums provided in the CUB library [89]. We could not find a high-performance GPU implementation of Select, hence, we instead use the implementation of RadixSort provided in the CUB library (which trivially allows us to find the k-th smallest priority in an array after it is sorted). These parallel subroutines are called from each thread block using CUDA dynamic parallelism.

The parBucketHeap is used to solve the SSSP problem using a parallel variant of Dijkstra's algorithm, denoted parDijsktra. Given a graph with n total vertices and maximum degree d, we perform n rounds where in each round: (1) the minimum distance vertex, denoted u, is extracted from the parBucketHeap; (2) all outgoing edges (u, v) are relaxed in parallel; and (3) all edges (u, v) that resulted in a shorter distance to v are inserted into the parBucketHeap via a BulkUpdate operation. This algorithm can be implemented using parallel scans and PrefixSums (on a maximum of d elements). To reduce the number of BulkUpdate operations, we set the maximum update batch size to be equal to d.

3.5.2 Methodology

We experimentally compare the performance of parDijsktra to the state-of-the-art SSSP GPU implementations, Gunrock [107] and Asynchronous Dynamic Delta-Stepping (ADDS)[5] [105], both of which are GPU implementations of the Delta-stepping algorithm. Meyer et al. [80] proved that for an arbitrary graph with random edge weights, the performance of the Delta-stepping algorithm is a function of the maximum degree d and maximum shortest path length L of the input graph. Moreover, past experimental work [17, 31] showed that the performance of previous implementations of Delta-stepping on GPUs degrade on sufficiently dense graphs with large diameters (i.e., the number of edges in the maximum shortest path). In this work, we demonstrate that using the parallel cache-efficient parBucketHeap in a parallel variant of Dijkstra's algorithm provides a suitable implementation for solving SSSP on GPUs for such graphs. Therefore, our experiments are conducted on graphs with sufficiently large degrees and diameter.

We generate 5 random directed acyclic graphs (DAGs) containing $n = 30{,}000$ vertices with diameter and maximum shortest path distance of $L = n - 1$. For 1-indexed vertices (i.e., vertices

[5]The ADDS library does not support compute capability 5.2

are identified via integers 1, 2, ..., n), we generate the random graphs with $m \geq n-1$ edges in two stages. In the first stage, the shortest paths (and diameter) are created such that for each $i \in \{1, 2, \ldots, n-1\}$, edge $(i, i+1)$ with weight 1 is inserted into the graph. Afterwards, the remaining $(m-n+1)$ edges are generated randomly with the following constraints: edges are distributed uniformly across the vertices $i \in \{1, 2, \ldots, n-1\}$, such that vertex i has a maximum degree of $(n-i)$; and if edge (u, v) is generated, then its weight is set to $(2 \cdot (u-v))$ to ensure that the shortest paths (and diameter) of the graph remains unchanged. Using these generated random graphs, the depth and work of the Delta-stepping algorithm becomes $O(dn \log n + \log^2 n)$ and $O(n+m+dn \log n)$, respectively, in the average case. In comparison, parDijsktra (using the parBucketHeap) has a parallel depth of $O(n(1+\log d))$ and $O\left(n+m+dn \log \frac{n}{d}\right)$ work, when running on the generated random graphs.

All code is compiled using CUDA C/C++ 11 and experiments are performed on 2 NVIDIA GPUs: an RTX 2080 Ti (compute capability 7.5), containing 4,352 physical processors distributed across 68 SMs, 11 GB of global memory, and 96 KiB of unified L1 cache and shared memory per SM; and a Quadro M4000 (compute capability 5.2), containing 1,664 physical processors distributed across 13 SMs, 8 GB of global memory, and 96 KiB of shared memory per SM. All runtime experiments are conducted on each of the generated input graphs, where for each graph, 10 trials are conducted. The average runtime across all trials (for all input graphs) are reported.

3.5.3 Runtime Results

Figure 3.5.3 and Figure 3.5.3 plots the average runtime of each of the SSSP algorithms across all generated random DAGs, compared to the number of edges in the input graph. Since edges are distributed uniformly in the generated random DAGs, as the number of edges increase, the maximum (and average) degree of the input graph also increases. Additionally, as the impact of I/O efficiency on overall runtime is pronounced on large input sizes (i.e., a large number of edges), we expect the parDijsktra using the I/O efficient parBucketHeap to perform well on these inputs.

Results show that on the RTX 2080 Ti, parDijsktra is faster than Gunrock and ADDS once the number of edges in the input graph exceeds 80 million edges and 100 million edges, respectively. And on the Quadro M4000, parDijsktra is faster than Gunrock once the the number of edges in the input graph exceeds 40 million edges. Furthermore, we find that while the ADDS implementation is faster than Gunrock for less than 80 million edges, its performance degrades significantly compared to Gunrock for graphs with a larger number of edges. On the RTX 2080 Ti, we observe a peak speedup of 2.8 compared to Gunrock and a peak speedup of 12 compared to ADDS, on the graphs with 300 million edges and 450 million edges, respectively. The peak speedup of parDijsktra compared to Gunrock on the Quadro M4000 is 5.4, occurring at 200 million edges. We note that on the Quadro M4000, Gunrock was unable to run on the graphs with 450 million

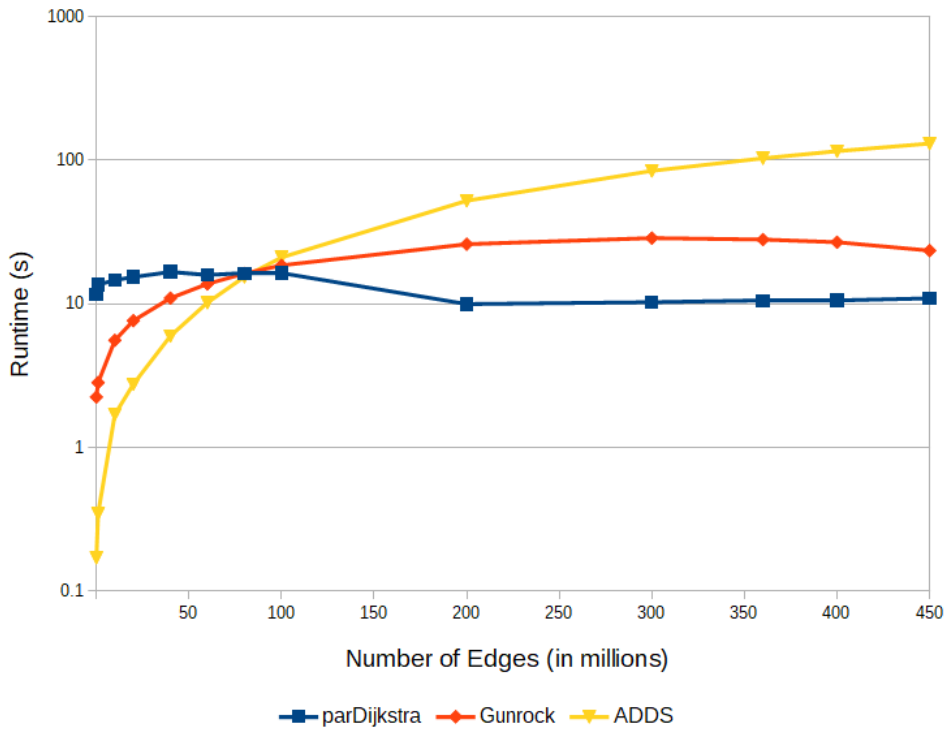

Figure 3.3: Average runtime (in seconds) on the generated random DAGs with $n = 30$ thousand vertices and diameter $L = n$ on a RTX 2080 Ti.

edges, due to an out-of-memory error.

3.6 Conclusion

In this chapter, we present the parallel bucket heap, denoted parBucketHeap, a parallel variant of the cache-oblivious bucket heap [15] and buffer heap [22]. The parallel bucket heap supports standard priority queue operations: Update, Delete, and ExtractMin, as well as BulkUpdate of up to d elements. For a maximum of n elements in the parBucketHeap, all operations can be performed with an amortized depth of $O(1 + \log d)$ and $O(d \log_4 \frac{n}{d})$ work in the CREW PRAM model. To optimize for I/O efficiency, d is bounded by the internal memory size of a processor (i.e., $d = O(M)$), resulting in $O\left(\frac{\log_4 n/d}{pB} + \frac{1}{B}\right)$ amortized parallel I/Os per Update, Delete, or ExtractMin operation; and $O\left(\frac{d}{pB} \cdot \log_4 \frac{n}{d} + \frac{d}{B}\right)$ amortized parallel I/Os per BulkUpdate operation, in the PEM model.

We implement the parBucketHeap on the GPU using CUDA C/C++ and use it in a parallel variant of Dijkstra's algorithm for solving the SSSP problem, denoted parDijsktra. Experimental results show that on an NVIDIA RTX 2080 Ti and Quadro M4000, the parDijsktra outperforms the current state-of-the-art SSSP GPU implementations, Gunrock [107] and ADDS [105], on our

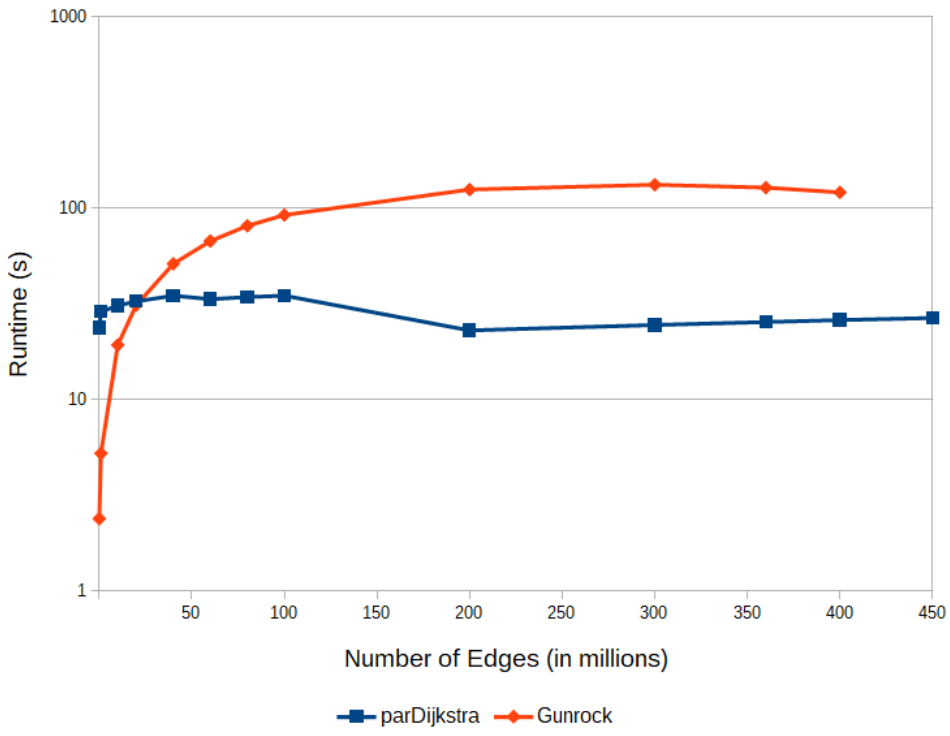

Figure 3.4: Average runtime (in seconds) on the generated random DAGs with $n = 30$ thousand vertices and diameter $L = n$ on a Quadro M4000.

generated random DAGs with $n = 30,000$ vertices and diameter $n - 1$. On the RTX 2080 Ti, we observe a peak speed up of 2.8 and 12 compared to Gunrock and ADDS, respectively; and on the Quadro M4000, parDijsktra provides a peak speed up of 5.4 compared to Gunrock. This work highlights the unique architecture of GPUs and how the thread and memory hierarchy can be leveraged to obtain both I/O efficiency per thread block and a high degree of parallelism.

CHAPTER 4
WORST-CASE INPUTS FOR PAIRWISE MERGE SORT

The current fastest comparison-based sorting implementation on GPUs is a parallel pairwise merge sort algorithm (provided in the Thrust library [88]). To achieve fast runtimes, the number of threads t to sort the input of N elements is fine-tuned experimentally for each generation of NVIDIA GPUs in such a way that the number of elements $E = N/t$ that each thread accesses in each merging round results in a small (empirically measured) number of bank conflicts in shared memory, while balancing the number of global memory accesses and latency-hiding through thread oversubscription/occupancy.

While some metrics, such as global memory accesses, can significantly dominate the overall performance of an algorithm (due to its larger latency compared to other memory accesses), for algorithms that heavily utilize other levels of the memory hierarchy, the corresponding metrics cannot always be ignored nor should be hidden in the asymptotic analysis. In particular, it has been shown that for some algorithms, there is a strong correlation between the number of bank conflicts in the shared memory and the overall runtime of GPU implementations [54, 67, 98].

Unfortunately, analyzing bank conflicts in shared memory can be difficult, especially for algorithms with data-dependent accesses. One approach taken in the past is to design algorithms that eliminate bank conflicts altogether, known as *bank conflict free* algorithms [1, 20, 66, 68, 98]. As a simple example, Dotsenko et. al [36] observed that for a simple parallel scan it is sufficient to pad the input in shared memory, such that the number of elements that each thread scans is co-prime with the total number of memory banks. In general, the following result is easy to prove using the pigeonhole principle:

Lemma 26. *Consider a warp of w threads accessing data stored in k consecutive addresses of memory organized into w memory banks, such that bank i contains all addresses $x \equiv i \pmod{w}$. Then there is a set of w (distinct) addresses, access to which will result in $\min\left\{\left\lceil \frac{k}{w} \right\rceil, w\right\}$ bank conflicts.*

Lemma 26 provides a simple worst-case bound on the number of bank conflicts for every parallel access to shared memory. However, depending on the particular algorithm's access pattern in shared memory, this bound may be too pessimistic. In particular, the above result does not consider any dependence between various accesses, which could preclude simultaneous access to the set of addresses defined by Lemma 26. Instead, a tighter analysis of specific algorithms needs to be performed to accurately analyze its performance in shared memory.

The work presented in this chapter takes a step in the direction of addressing this lack of theoretical analysis of bank conflicts in existing algorithms by showing that in the case of the GPU pairwise merge sort algorithm, the bound of Lemma 26 is indeed asymptotically tight. We show

that for every choice of $E \leq w$, there exists an input permutation on which every warp of w threads of the pairwise merge sort algorithm is effectively reduced to using at most $\lceil w/E \rceil$ threads due to sequentialization of shared memory accesses due to bank conflicts. Our proof is constructive, i.e., we are able to automatically construct such permutation for every value of E. Additionally, we show that such constructed inputs result in up to $\approx 50\%$ slowdown, compared to the performance on random inputs, on modern GPU hardware.

4.1 Preliminaries

4.1.1 GPU Pairwise Merge Sort

The GPU pairwise merge sort algorithm is based on the GPU Merge Path algorithm [55], which is a high-performance implementation of pairwise merging on a GPU. Let A and B be two sorted lists such that $|A| + |B| = n$ and let t be the total number of threads. GPU Merge Path is divided into two stages: a partitioning stage and a merging stage. The idea of the partitioning stage is to identify for each thread $i \in \{1, 2, ..., t\}$ the i-th quantile (i-th group of n/t smallest elements) to be merged by the i-th thread during the merging stage independently of other threads. By using the order-statistics of two sorted lists (via mutual binary search), each thread is able to compute the starting location of its quantile in the A and B lists. Then, in the merging stage, each thread performs a sequential merge of n/t elements independently of other threads.

Let N be the number of elements and w be the number of threads per warp. The GPU pairwise merge sort implementation uses the following tuning parameters, which are chosen empirically: b is the number of threads per thread block and E is the number of elements that each thread will work on in each merging round, i.e., the total number of threads is chosen to be N/E. The parameter b is chosen to be a power of two. The algorithm starts with the base case where chunks of bE consecutive elements are sorted in shared memory in parallel using b threads per chunk, i.e., each thread block sorts an independent partition of bE elements. In order to do this, each thread first sorts E elements in registers via an odd-even sorting network [56]. Then, each thread block performs a pairwise merge sort using $\log b$ merge rounds, where in each round $i \in \{1, 2, ..., \log b\}$, $(b/2^i)$ pairs of lists, each of size $2^{i-1}E$, are merged via GPU Merge Path using 2^i threads.

Once each chunk of bE elements is sorted, $\lceil \log \frac{N}{bE} \rceil$ pairwise merge rounds are performed, where in each round $i \in \{1, 2, ..., \lceil \log \frac{N}{bE} \rceil\}$, 2^i thread blocks work together to perform a pairwise merge on $2^{i-1}bE$ elements per list. Thus, each thread block needs to find its quantile of bE elements in the two sorted lists. These elements will then be merged by the thread block in shared memory, independently of other thread blocks. Similar to GPU Merge Path, each thread block computes the starting addresses of its quantile in the two sorted lists via a mutual binary search in global memory. Then, the thread block proceeds by performing a single round of GPU Merge Path in shared memory on its bE elements.

4.1.2 Related Work

Over the years, various sorting implementations for GPUs have been developed such as: pairwise merge sort [7, 55, 88, 96, 98], multiway merge sort [67, 72], multiway distribution sort [73], shear sort [1, 98], bitonic sort [90, 91] and radix sort [89, 96]. Recent empirical studies have shown that the current state-of-the-art comparison-based sorting implementation on GPUs is the pairwise merge sort implementations available in the Thrust and Modern GPU libraries [67, 79].

Experimental results show that Thrust and Modern GPU perform well on random inputs [67, 79]. However, typical experiments are performed on at most a dozen random inputs with the average runtime reported (often without any mention of variance or other statistics). For the problem of comparison-based sorting, out of $n!$ possible permutations, a random sample of only a dozen inputs represents no statistical significance.

Karsin *et al.* [65, 67] perform theoretical analysis of the GPU pairwise merge sort algorithm by computing the number of parallel coalesced accesses in global memory, denoted A_g, and the number of parallel shared memory accesses (with the number of bank conflicts parameterized), denoted A_s. Let P be the number of physical cores on the GPU, β_1 be the average number of bank conflicts per iteration of the mutual binary search (i.e., the partitioning stage), and β_2 be the average number of bank conflicts per iteration of merging (i.e., merging stage). Then

$$A_g = O\left(\frac{Nw}{PbE}\log^2\left(\frac{N}{bE}\right) + \frac{N}{P}\log\frac{N}{bE}\right)$$

$$A_s = O\left(\frac{N}{PE}\log\left(\frac{N}{bE}\right)(\beta_1 \log bE + \beta_2 E)\right)$$

Karsin *et al.* [67] found empirically that for Modern GPU on random inputs, $\beta_1 = 3.1$ and $\beta_2 = 2.2$. The authors show a strong correlation between the number of bank conflicts and the runtime of Modern GPU for a fixed input size of 10^8 integer elements. Furthermore, so-called *conflict-heavy* inputs are constructed (i.e., inputs that cause a "large" number of bank conflicts) and showed that these inputs increase the runtime of Modern GPU and Thrust, compared to random inputs. Unfortunately, these conflict-heavy inputs were constructed manually for two specific software configuration parameters, the comparison of these conflict-heavy inputs with random inputs is only shown for the GTX 770 (compute capability 3.0), and theoretical analysis of the number of bank conflicts incurred was not investigated and was left as an open problem. This work addresses this open problem.

4.1.3 Our approach

Observe that in the bound for the number of parallel shared memory accesses (A_s) in Section 4.1.1, the number of accesses in the merging stage is larger than the number of accesses in the partitioning stage when $E \geq \log bE$. In practice, this inequality is satisfied for all values of E and b used in

Thrust and Modern GPU [7, 88]. Therefore, we focus on constructing the worst-case input for the merging stage.

We consider the *pairwise merge problem*, where two sorted lists A and B, each of size $\frac{bE}{2}$, are being merged using b threads of the same thread block. We assume that each thread knows the addresses within A and B from where it will start the merging process (i.e., the partitioning stage of GPU Merge Path has been performed) and it will read the E elements that it is assigned in the increasing order of their values.

Memory alignment

To simplify our task, we restrict our attention to a simpler problem: maximizing bank conflicts that occur within a fixed set of E *contiguous* memory banks. This is equivalent to a simpler problem of finding a permutation that maximizes the number of threads synchronously scanning elements that are located on the chosen E consecutive memory banks. Since we are generating a worst-case input, this restriction only strengthens our result.

Let $s \in \mathbb{Z}_w$ be the starting memory bank of the chosen E consecutive memory banks.[1] Since execution within a warp is performed in lock-step, we view each merging round as an execution of E steps or, equivalently, E accesses to shared memory. We say an input element e is *aligned* (with respect to the E banks), if e is read in time step $j \in \mathbb{Z}_E$ and is located in bank $(s + j) \pmod w$. Since a thread is reading its E elements in increasing order of the values, the alignment is simply a function of the relative order among the E elements being merged by the thread.

General Strategy

Recall from Section 4.1.1 that in each merge round, each thread block finds its partition of bE elements to merge across 2 sorted lists, A and B; and each thread finds its E elements to merge, within the bE elements of its corresponding thread block. Thus, when constructing our worst-case input, we have the freedom to choose the number of elements in the A and B lists for a thread block, as long as the total number of elements within a thread block equals to bE and the total number of elements in the A and B lists across all thread blocks is equal. Additionally, within a thread block, we have the freedom to choose the number of elements in the A and B lists assigned to a warp, as long as the total number of elements in the A and B lists across all warps in the thread block is consistent with the total number of elements given to that particular thread block.

In order to consider each thread block independently, we design our input so that each thread block is always given $\frac{bE}{2}$ elements from both the A and B lists. For each warp, we decide to give $\lceil \frac{E}{2} \rceil w$ elements in one list and $\lfloor \frac{E}{2} \rfloor w$ elements in the other list. This choice allows us to fix the start of the A and B lists for each warp to the 0-th memory bank, as well as allow us to consider each warp independently (without loss of generalization).

[1] For any integer $c \geq 1$, $\mathbb{Z}_c = \{0, 1, 2, ..., c-1\}$.

Formally, we partition a thread block into 2 disjoint sets L and R, such that L and R both contain $\frac{b}{2w}$ warps. For every warp $l \in L$, we assign $\lceil \frac{E}{2} \rceil w$ elements of the A list and $\lfloor \frac{E}{2} \rfloor w$ elements of the B list. And for warps $r \in R$, we perform the symmetric assignment of $\lfloor \frac{E}{2} \rfloor w$ elements of the A list and $\lceil \frac{E}{2} \rceil w$ elements of the B list.

Ideally, our goal is to generate an input for each warp such that E threads perform a scan of E consecutive elements starting at memory bank s. Therefore, we design our input such that every thread performs a scan of one list then the other list, i.e., for some integer $0 \leq k \leq E$, the first k elements merged belong to one list and the remaining $(E - k)$ elements belong to the other list. Furthermore, our inputs are generated with a strategy that ensures that for each thread, only elements from a single list will start within the E consecutive banks, which makes it clear which list to scan first. Thus, we can indirectly describe our input by assigning the number of elements in each list that a particular thread reads.

4.2 Worst-Case Bank Conflict Analysis

Let $1 < E \leq w$ and $d = \text{GCD}(w, E)$. We consider an arbitrary warp in the merging stage where $n = wE$ elements reside in shared memory and each thread has performed merge path to find its independent partition of E elements to merge. Without loss of generality, we assume that $|A| = \lceil \frac{E}{2} \rceil w$ and $|B| = \lfloor \frac{E}{2} \rfloor w$. We partition the n elements into d subproblems of $n' = \frac{wE}{d}$ elements. Without loss of generality, we consider $\frac{w}{d}$ threads, $\lceil \frac{E}{2d} \rceil w$ elements of A, and $\lfloor \frac{E}{2d} \rfloor w$ elements of B.

From Euclid's Division Lemma 36, there exists unique positive integers q and r such that $0 \leq r < E$ and $w = qE + r$. For $i = 1, 2, \ldots, \frac{E}{d} - 1$, let

$$
\begin{aligned}
s_i &= i\left(\frac{r}{d}\right) \pmod{E/d} \\
x_i &= \left(\frac{E}{d} - s_i\right) d \\
y_i &= s_i \cdot d
\end{aligned}
$$

and let $S = (a_i, b_i)$ be a sequence such that,

$$
a_i = \begin{cases} x_i & \text{if } i \text{ is even} \\ y_i & \text{otherwise} \end{cases}
$$

$$
b_i = \begin{cases} x_i & \text{if } i \text{ is odd} \\ y_i & \text{otherwise.} \end{cases}
$$

We note that from Lemma 43,

$$d = \text{GCD}(w, E) = \text{GCD}(E, r)$$

and from Lemma 44,

$$\begin{aligned} \text{GCD}\left(\frac{w}{d}, \frac{E}{d}\right) &= 1 \\ \text{GCD}\left(\frac{E}{d}, \frac{r}{d}\right) &= 1 \ . \end{aligned}$$

Lemma 27. *For any $i, j \in \{1, 2, \ldots, \frac{E}{d} - 1\}$ such that $i \neq j$,*

$$s_i \neq s_j \ .$$

Proof. From Lemma 53, there exists a unique inverse for $\left(\frac{r}{d}\right)$ modulo $\left(\frac{E}{d}\right)$, denoted $\left(\frac{r}{d}\right)^{-1}$. Assume, for the sake of contradiction, that $s_i = s_j$.

$$\begin{aligned} &\implies i\left(\frac{r}{d}\right) \equiv j\left(\frac{r}{d}\right) \pmod{E/d} \\ \implies &i\left(\frac{r}{d}\right)\left(\frac{r}{d}\right)^{-1} \equiv j\left(\frac{r}{d}\right)\left(\frac{r}{d}\right)^{-1} \pmod{E/d} \\ &\implies i \equiv j \pmod{E/d} \\ &\implies i = j \ . \end{aligned}$$

A contradiction. Hence, $s_i \neq s_j$. □

Lemma 28. *For $i \in \{1, 2, \ldots, \frac{E}{d} - 1\}$,*

$$\frac{E}{d} - s_i = s_{\frac{E}{d} - i} \ .$$

Proof.

$$\begin{aligned} \frac{E}{d} - s_i &= \frac{E}{d} - \left(i\left(\frac{r}{d}\right) \pmod{E/d}\right) \\ &\equiv \left(\frac{E}{d} - i\right)\left(\frac{r}{d}\right) \pmod{E/d} \\ &= s_{\frac{E}{d} - i} \ . \end{aligned}$$

□

Lemma 29. *For* $i = 1, 2, \ldots, \frac{E}{d} - 2$

$$x_i + y_{i+1} = \begin{cases} r & \text{if } x_i < r \iff s_i > \frac{E}{d} - \frac{r}{d} \\ E + r & \text{otherwise} \end{cases}$$

Proof. (Note that $x_i = r \iff i = \frac{E}{d} = 1$, therefore, there are only two cases.)

Case 1: Assume $x_i < r \iff s_i > \frac{E}{d} - \frac{r}{d}$.

$$\begin{aligned} &\implies s_i + \frac{r}{d} > \frac{E}{d} \\ &\implies s_{i+1} = s_i + \frac{r}{d} - \frac{E}{d} \ . \end{aligned}$$

Therefore,

$$\begin{aligned} x_i + y_{i+1} &= \left(\frac{E}{d} - s_i \right) d + s_{i+1} \cdot d \\ &= \left(\frac{E}{d} - s_i + s_i + \frac{r}{d} - \frac{E}{d} \right) d \\ &= r \ . \end{aligned}$$

Case 2: Assume $x_i > r \iff s_i < \frac{E}{d} - \frac{r}{d}$.

$$\begin{aligned} &\implies s_i + \frac{r}{d} < \frac{E}{d} \\ &\implies s_{i+1} = s_i + \frac{r}{d} \ . \end{aligned}$$

Therefore,

$$\begin{aligned} x_i + y_{i+1} &= \left(\frac{E}{d} - s_i \right) d + s_{i+1} \cdot d \\ &= \left(\frac{E}{d} - s_i + s_i + \frac{r}{d} \right) d \\ &= E + r \ . \end{aligned}$$

□

Intuitively, the goal is to align each column of w elements so that the number of elements that are assigned to the threads correspond to the structure of Euclid's Division Lemma 36 for w and E,

$$w = qE + r \ .$$

The sequence S provides the order of elements to assign to threads to pad the first r or $(E + r)$

elements in each column. Afterwards, q or $(q-1)$ scans of E elements can be performed. Formally, we construct a new sequence T from S as follows:

1. Insert q tuples of $(E, 0)$ after $(a_1, b_1) = (y_1, x_1) = (r, E - r)$
2. For $i = 1, 2, \ldots, \frac{E}{d} - 2$, after every pair (a_{i+1}, b_{i+1}) such that $x_i + y_{i+1} = r$, insert q tuples

$$\begin{cases} (E, 0) & \text{if } i \text{ is even} \\ (0, E) & \text{otherwise.} \end{cases}$$

   Otherwise, after every pair (a_{i+1}, b_{i+1}) such that $x_i + y_{i+1} = E + r$, insert $(q-1)$ tuples

$$\begin{cases} (E, 0) & \text{if } i \text{ is even} \\ (0, E) & \text{otherwise.} \end{cases}$$

3. After $(a_{\frac{E}{d}-1}, b_{\frac{E}{d}-1})$, insert q tuples of $(E, 0)$ if $\frac{E}{d} - 1$ is even; otherwise insert q tuples of $(0, E)$.

In total, we have inserted $2q + q\left(\frac{r}{d} - 1\right) + (q-1)\left(\frac{E}{d} - \frac{r}{d} - 1\right)$ tuples and hence,

$$\begin{aligned} |T| &= \left(\frac{E}{d} - 1\right) + q\left(\frac{r}{d} + 1\right) + (q-1)\left(\frac{E}{d} - \frac{r}{d} - 1\right) \\ &= \left(\frac{E}{d} - 1 - \frac{E}{d} + \frac{r}{d} + 1\right) + q\left(\frac{r}{d} + 1 + \frac{E}{d} - \frac{r}{d} - 1\right) \\ &= \frac{r}{d} + q\left(\frac{E}{d}\right) = \frac{w}{d}\,. \end{aligned}$$

Theorem 30. *Using the sequence T to assign elements to the $\frac{w}{d}$ threads in the subproblem of $n' = \frac{wE}{d}$ elements, we can align accesses that result in a total of*

$$\begin{cases} \frac{E^2}{d} & \text{bank conflicts, if } 1 < E \leq \frac{w}{2} \\ \frac{1}{2}\left(\frac{E^2}{d} + \frac{2Er}{d} + E - \frac{r^2}{d} - r\right) & \text{bank conflicts, otherwise.} \end{cases}$$

Proof. Case 1: $1 < E \leq \frac{w}{2} \iff q > 1$.
Since $q > 1$, every column ends with (at least) a single scan of E elements (in particular, in shared memory banks $w-E, w-E+1, \ldots, w-1$), thus, resulting in a total of $E\left(\frac{E}{d}\right) = \frac{E^2}{d}$ bank conflicts.

Case 2: $\frac{w}{2} < E \leq w \iff q = 1$.
From Lemma 29, we know that there are $\left(\frac{E}{d} - 1 - \left(\frac{E}{d} - \frac{r}{d}\right)\right) = \left(\frac{r}{d} - 1\right)$ pairs that sum up to r and $\left(\frac{E}{d} - \frac{r}{d} - 1\right)$ pairs that sum up to $E + r = w$ Hence, $\left(\frac{r}{d} - 1\right)$ columns end with a single scan of E elements; and $\left(\frac{E}{d} - \frac{r}{d} - 1\right)$ columns end with a partial scan of elements.

Recall that for $i = 1, 2, \ldots, \frac{E}{d} - 2$, $x_i + y_{i+1} = E + r = w$ if $x_i > r \iff s_i < \frac{E}{d} - \frac{r}{d}$. Thus, $(x_i - r)$ elements are misaligned in the corresponding column. In total, there are

$$\begin{aligned}
\sum_{i=1}^{\frac{E}{d}-\frac{r}{d}-1} \left(\left(\frac{E}{d} - s_i\right) d - r\right) &= \sum_{i=1}^{\frac{E}{d}-\frac{r}{d}-1} \left(\left(\frac{E}{d} - \frac{r}{d} - s_i\right) d\right) \\
&= d + 2d + 3d + \ldots + \left(\frac{E}{d} - \frac{r}{d} - 1\right) d \text{ , since } 0 < s_i < \frac{E}{d} - \frac{r}{d} \\
&= d \sum_{i=1}^{\frac{E}{d}-\frac{r}{d}-1} i \\
&= d \left(\frac{\left(\frac{E}{d} - \frac{r}{d}\right)\left(\frac{E}{d} - \frac{r}{d} - 1\right)}{2}\right) \\
&= \left(\frac{E - r}{2}\right)\left(\frac{E}{d} - \frac{r}{d} - 1\right) \\
&= \frac{1}{2}\left(\frac{E^2}{d} - \frac{Er}{d} - E - \frac{Er}{d} + \frac{r^2}{d} + r\right) \\
&= \frac{1}{2}\left(\frac{E^2}{d} - \frac{2Er}{d} - E + \frac{r^2}{d} + r\right) \text{ misaligned elements.}
\end{aligned}$$

Therefore, there are a total of

$$\begin{aligned}
&\frac{E^2}{d} - \frac{1}{2}\left(\frac{E^2}{d} - \frac{2Er}{d} - E + \frac{r^2}{d} + r\right) \\
&= \frac{1}{2}\left(\frac{E^2}{d} + \frac{2Er}{d} + E - \frac{r^2}{d} - r\right) \text{ bank conflicts.}
\end{aligned}$$

(We note that for $d = E \implies r = 0$, we misalign $\frac{1}{2}\left(\frac{E^2}{d} - \frac{2Er}{d} - E + \frac{r^2}{d} + r\right) = \frac{1}{2}(E - E) = 0$ elements.) □

For the symmetric case, where the subproblem of $n' = \frac{wE}{d}$ elements contains $\left\lfloor \frac{E}{2d} \right\rfloor w$ elements of A and $\left\lceil \frac{E}{2d} \right\rceil w$ elements of B, the symmetric strategy is used where the sequence T is used with tuple values switched. Therefore, combining all d subproblems together results in a total of

$$\begin{cases}
E^2 & \text{bank conflicts, if } 1 < E \leq \frac{w}{2} \\
\frac{1}{2}\left(E^2 + 2Er + Ed - r^2 - rd\right) & \text{bank conflicts, otherwise.}
\end{cases}$$

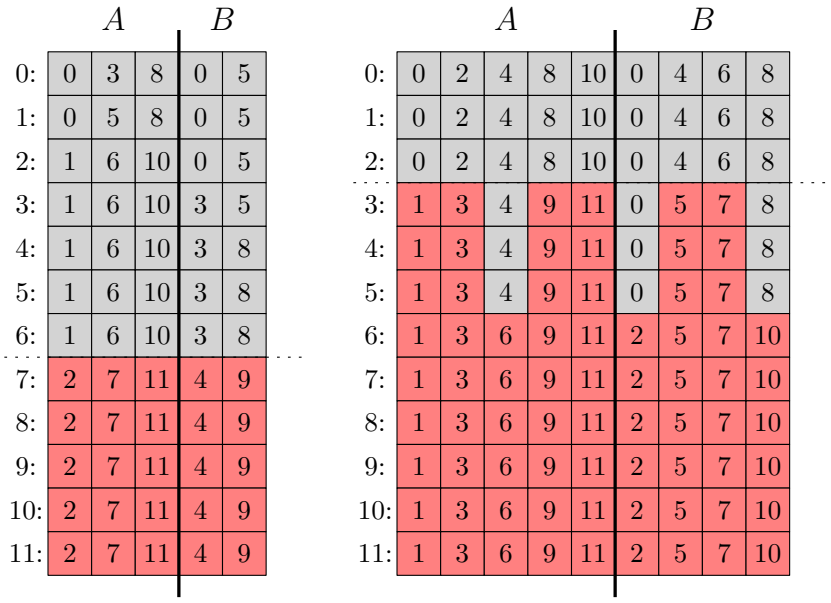

Figure 4.1: Visualization of the constructed worst case inputs for a single warp for $w = 12$ threads per warp. The left figure shows the inputs for $E = 5$ (i.e., coprime) and the right shows $E = 9$ (i.e., not coprime). Cells are numbered with the thread that accesses the element. In both figures, the elements residing in the last E memory banks are aligned to cause bank conflicts when threads load its subsequences into registers. Red cells correspond to the accesses that contribute to the worst case number of bank conflicts.

4.3 Experimental Results

4.3.1 Methodology

We perform our experiments on 2 NVIDIA GPUs: a Quadro M4000 (compute capability 5.2), which contains 1664 physical processors across 13 SM's, 8 GB of global memory, and 96 KiB of shared memory per SM; and an RTX 2080 Ti (compute capability 7.5), which contains 4352 physical processors across 68 SM's, 11 GB of global memory, and 96 KiB of unified L1 cache and shared memory per SM (configured at runtime to be either 32 KiB of L1 cache and 64 KiB of shared memory, or vice versa)[2].

We use the Thrust library included with the CUDA 10.1 toolkit, which defines the software parameters of $E = 15$ and $b = 512$ for the Quadro M4000. However, the software parameters are not explicitly defined for the RTX 2080 Ti and by default it uses the parameters defined for compute capability 6.0, which is $E = 17$ and $b = 256$. For these software parameters, each thread block requires 17 KiB of shared memory space, thus, 3 thread blocks (768 total threads) using a total of 51 KiB of shared memory space (13 KiB unused) can be resident on each SM. Compared to $E = 15$ and $b = 512$, each thread block uses 30 KiB of shared memory space, which results in

[2]GB = 10^9 B and KiB = 2^{10} B

2 resident thread blocks (1024 total threads) using a total of 60 KiB of shared memory space (4 KiB unused). As the RTX 2080 Ti can support up to 1024 resident threads per SM, the latter parameters provides 100% theoretical occupancy while the former parameters provides only 75% theoretical occupancy. Therefore, we expect $E = 15$ and $b = 512$ to outperform $E = 17$ and $b = 256$. In our experiments we use both of these parameters for the RTX 2080 Ti.

The Modern GPU library defines $E = 15$ and $b = 128$ for the Quadro M4000 and, similar to Thrust, does not explicitly define parameters for the RTX 2080 Ti. Hence, we run experiments using the same two sets of parameters as in our Thrust experiments.

All experiments are performed on 4-byte integers with the average over 10 runs being reported. Runtimes are recorded using `cudaEventRecord` and bank conflict counts are gathered via NVIDIA's provided profilers. Specifically, for the Quadro M4000, `nvprof` is used to record both `shared_ld_bank_conflict` and `shared_st_bank_conflict` with the sum of these two metrics providing the total number of bank conflicts; and for the RTX 2080 Ti, `nv-nsight-cu-cli` is used to record the `l1tex__data_bank_conflicts.sum` metric. The test harness program for Thrust is compiled with the `-O3` optimization flag and the test harness program for Modern GPU is compiled with its provided Makefile.

4.3.2 Results

Figure 4.2 shows both the Thrust and Modern GPU throughput results for their respectively defined software parameters on the Quadro M4000. We find that the constructed worst-case inputs cause a peak slowdown of 50.49% (occurring at 7,864,320 elements) and 33.82% (occurring at 62,914,560 elements) for Thrust and Modern GPU, respectively. Overall, we have an average slowdown of 43.53% and 27.3% for Thrust and Modern GPU, respectively. Moreover, as expected, Thrust outperforms Modern GPU for both random and constructed worst-case inputs.

Figure 4.3 and Figure 4.4 shows the throughput results for both software parameters in Thrust and Modern GPU, respectively, on the RTX 2080 Ti. We find that for $E = 15$ and $b = 512$, the constructed worst-case inputs cause a peak slowdown of 42.43% (occurring at 31,457,280 elements) and 42.62% (occurring at 3,932,160 elements) for Thrust and Modern GPU, respectively. The average slowdown is 33.31% and 35.25% for Thrust and Modern GPU, respectively. For $E = 17$ and $b = 256$, the constructed worst-case inputs cause a peak slowdown of 22.94% (occurring at 35,651,584 elements) and 20.34% (occurring at 285,212,672 elements) for Thrust and Modern GPU, respectively. The average slowdown is 16.54% and 12.97% for Thrust and Modern GPU, respectively.

On the RTX 2080 Ti, results from both Thrust and Modern GPU confirm that for random inputs, $E = 15$ and $b = 512$ provide increased performance over $E = 17$ and $b = 256$. However, it is interesting that for the constructed worst-case inputs, the opposite is true: $E = 17$ and $b = 256$ outperforms $E = 15$ and $b = 512$. This results in a much larger slowdown for $E = 15$ and $b = 512$

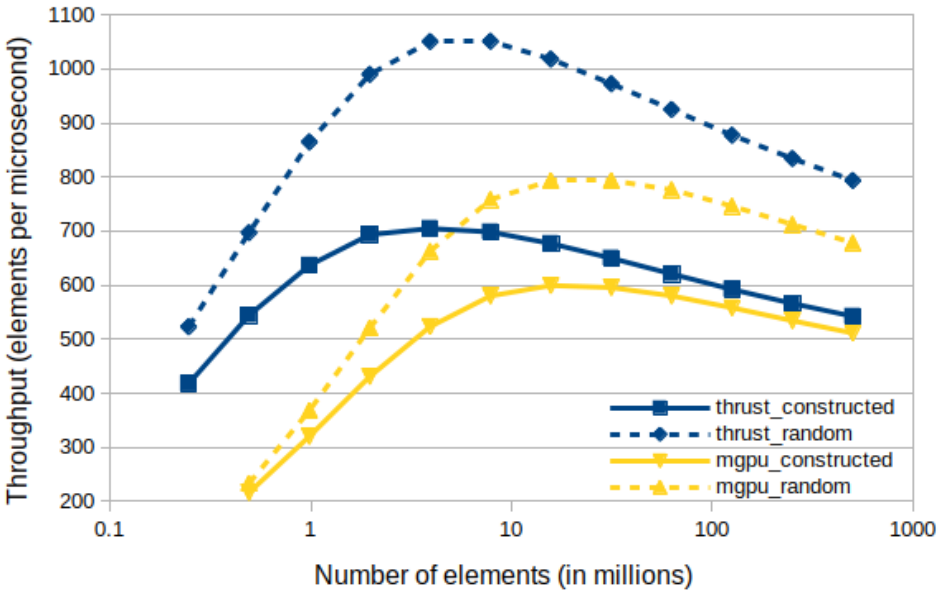

Figure 4.2: Throughput results for both Thrust and Modern GPU on the Quadro M4000. Thrust results are in blue and Modern GPU results are in yellow. The solid lines represent the constructed worst-case inputs and the dashed lines represent random inputs. The x-axis is on a logarithmic scale.

compared to $E = 17$ and $b = 256$. To investigate this, we compare the runtime per element and the bank conflicts per element for both software parameters (Figure 4.5 shows this comparison for Thrust). We find that the relative performance of the number of bank conflicts per element predicts the relative performance of the runtime per element. In other words, there is indeed a correlation between the runtime and the number of bank conflicts. Moreover, as we expect, the number of bank conflicts per element shows logarithmic growth; and while there is some noise from the base case, we also see logarithmic growth in the runtime per element.

4.4 Conclusion

In this chapter we showed that for every value of $1 < E \leq w$, there exists an input that reduces the effective parallelism of each warp on the GPU from w down to $\lceil w/E \rceil$ due to memory contention in shared memory. This translates into non-trivial slowdown on such inputs in practice. One natural question that might arise from this work is: the constructed worst-case input is a very specific permutation and, thus, is very unlikely to occur with high frequency in real world inputs. So why should we care about the worst-case performance? This is a philosophical question that can be addressed from several aspects:

1. Every undergraduate algorithms course teaches that we should analyze algorithm runtimes on

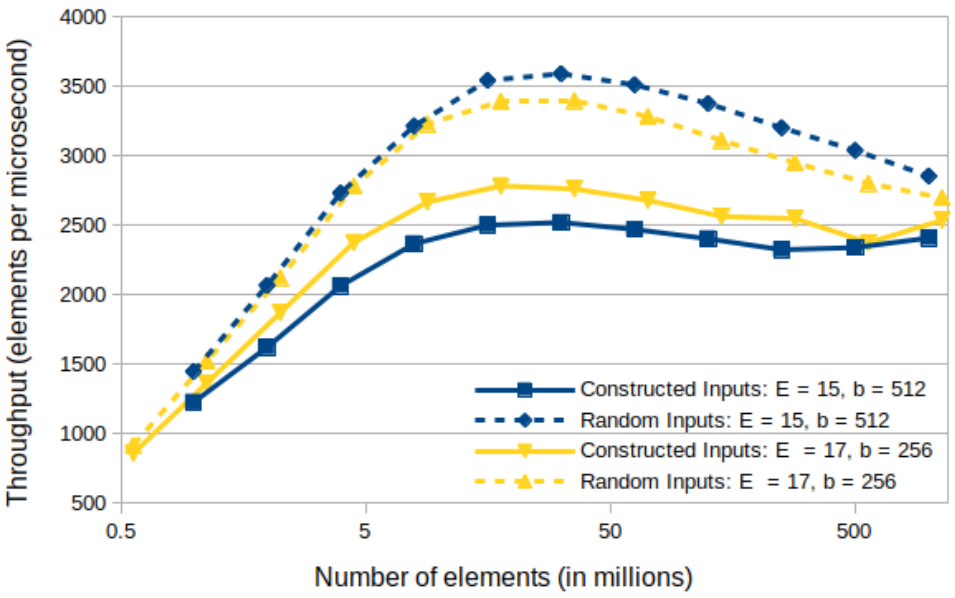

Figure 4.3: Throughput results for Thrust on the RTX 2080 Ti. The blue lines represent parameters $E = 15$ and $b = 512$ and the yellow lines represent the parameters $E = 17$ and $b = 256$. The solid lines represent the constructed worst-case inputs and the dashed lines represent random inputs. The x-axis is on a logarithmic scale.

the worst-case inputs. Why should we ignore such analysis for GPU algorithms? Moreover, such analysis might lead to the discovery of better algorithmic techniques on GPUs.

2. The goal of this paper was to prove the existence of a single permutation that asymptotically matches the pessimistic bound of Lemma 26 for the parallel pairwise merge sort algorithm. However, observe that our construction can actually produce a family of permutations, as many of the elements in the non-aligned $w - E$ memory banks can be permuted without affecting the total number of bank conflicts.

3. We could relax our requirement for an absolute worst-case and produce a permutation that has slightly fewer bank conflicts than our constructed permutation. Therefore, there are many more permutations that still incur a significant number of bank conflicts.

4. Observe that the runtimes on the worst-case inputs represent an extreme end of the possible runtime variance. With the constructed inputs causing an average slowdown of $\approx 43\%$ and $\approx 33\%$ on a Quadro M4000 and a RTX 2080 Ti, respectively, the possible variance in runtime is quite significant.

A better question to ask is: can we analyze the expected number of bank conflicts for a given algorithm, for a specific input distribution? This seems to be a very difficult problem for any non-

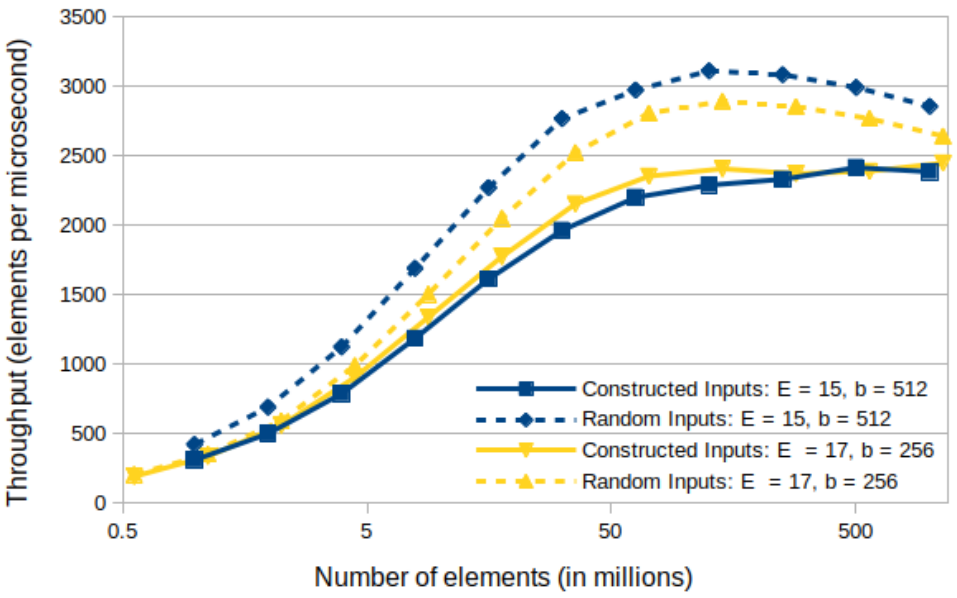

Figure 4.4: Throughput results for Modern GPU on the RTX 2080 Ti. The blue lines represent parameters $E = 15$ and $b = 512$ and the yellow lines represent the parameters $E = 17$ and $b = 256$. The solid lines represent the constructed worst-case inputs and the dashed lines represent random inputs. The x-axis is on a logarithmic scale.

trivial data-dependent algorithm. Understanding how such data dependencies can be modeled so we can apply standard randomized analysis techniques is an interesting open problem. We hope that the analysis presented here will act as the first step in this direction.

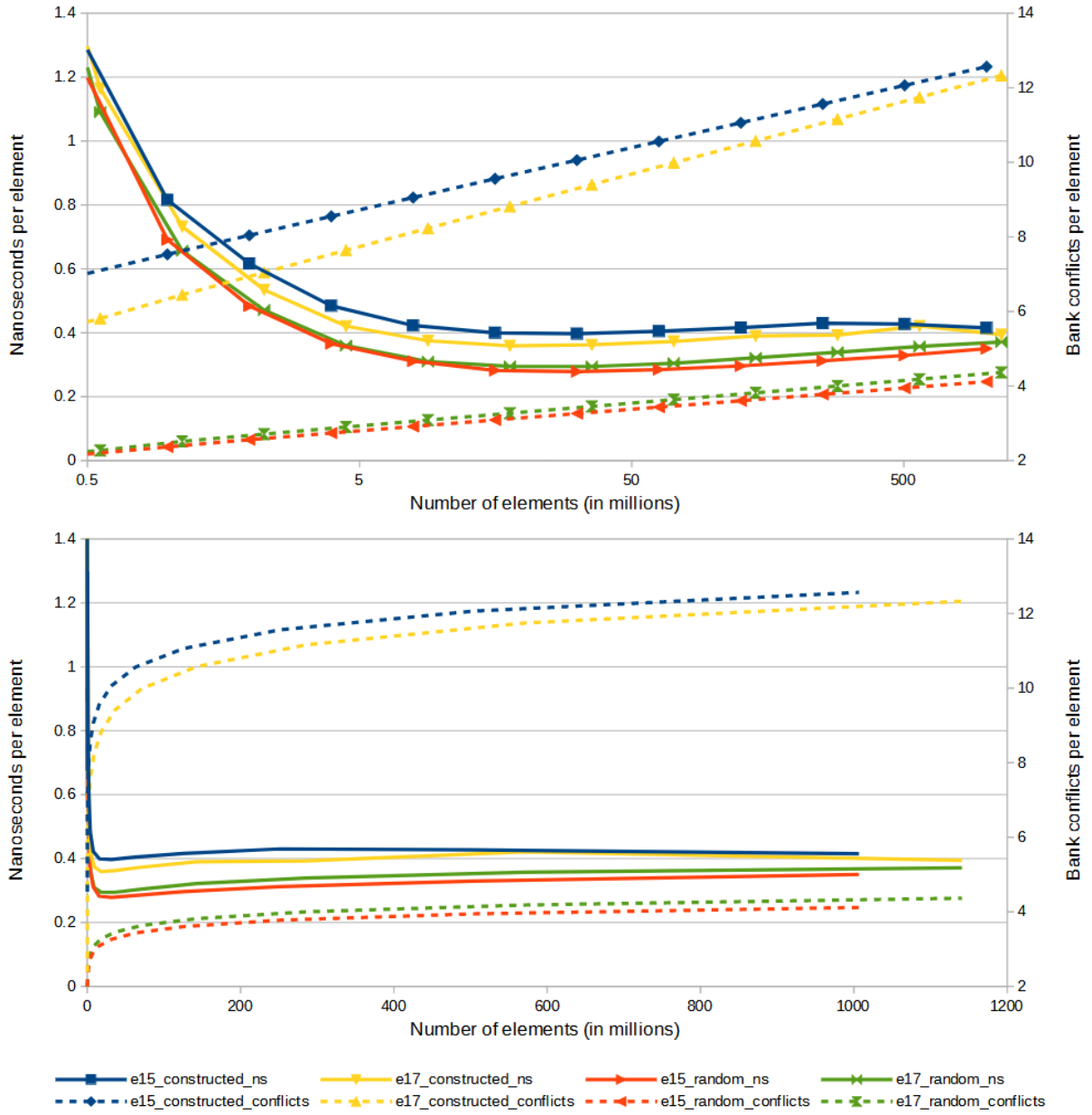

Figure 4.5: Runtime (in nanoseconds) per element and bank conflicts per element for Thrust on the RTX 2080 Ti. The top figure displays the data with the x-axis on a logarithmic scale, to clearly show each individual data point. While the bottom figure displays the data without data points shown and with the x-axis on a linear scale in order to emphasize the resulting logarithmic shape of the curves. In both figures, the solid lines represent the runtime (in nanoseconds) per element and the dashed lines represent the bank conflicts per element.

CHAPTER 5
BANK CONFLICT FREE DIVIDE-AND-CONQUER ALGORITHMS

In this chapter, we show that we can eliminate all bank conflicts for a large class of algorithms, known as *balanced two-way divide-and-conquer algorithms*, with virtually no overhead. Given an input array of size n, a typical (sequential) divide-and-conquer solution reduces the problem to two recursive calls on equal-sized subarrays and a scan of the array either to define the two subarrays or to combine the answers of the recursive call. If the scan of the array can be implemented efficiently in parallel in $T(n)$ time, such solutions are easily amenable to parallelization by executing the recursive calls in parallel resulting in $O(T(n) \log n)$ overall parallel time. For example, in case of the classical mergesort, given t threads, merging of two sorted arrays $\mathcal{A}$ and $\mathcal{B}$, each of size $n/2$, can be implemented in parallel by identifying t pairs of contiguous subarrays A_i and B_i, $|A_i| + |B_i| = n/t$, such that the sorted $A_i \cup B_i$ form a contiguous subarray in the sorted output of $\mathcal{A} \cup \mathcal{B}$. The identification of the i-th pair A_i and B_i is an order statistic that can be implemented by the i-th thread independently of other threads via a mutual binary search on $\mathcal{A}$ and $\mathcal{B}$ in $O(\log n)$ time and merging of A_i and B_i can be implemented by the i-th thread sequentially in $O(n/t)$ time. A large number of problems fall into this algorithmic framework, e.g., sorting algorithms (mergesort, quicksort), median and order statistics, plane sweep in 2 dimensions (e.g., convex hull, line segment intersections, closest pair), offline construction of augmented tree data structures (e.g., k-d trees, interval trees, segment trees), and many more.

In the context of GPUs, balanced two-way divide-and-conquer algorithms have been used to solve problems such as sorting [55], graph processing [31, 74], sparse matrix multiplication [77, 78, 110], list intersections [46], suffix array construction [106], and join operations for relational databases [94]. In these implementations, the identification of the i-th pair of subsequences A_i and B_i is an order statistic that can be found by the i-th thread independently of other threads via a mutual binary search on $\mathcal{A}$ and $\mathcal{B}$ in $O(\log n)$ time [55]. To reduce the number of global memory accesses, this procedure is performed in a 2-stage manner: first in global memory, where subsequences for each thread block are identified and loaded into contiguous shared memory locations; and secondly in shared memory, where each thread identifies A_i and B_i. Due to the data-dependent locations of A_i and B_i in shared memory, naive processing of these elements can result in bank conflicts. In general, analytically determining the number of bank conflicts on a random input in this setting is an open problem.

The current state-of-the-art implementation of mergesort on GPUs uses a heuristic of choosing t such that $|A_i| + |B_i| = n/t$ is coprime with w (the number of banks in shared memory) which they determined performs better in practice. Once A_i and B_i are identified in shared memory, the two subarrays are merged from shared memory into registers and transferred back into global

memory in sorted order. Karsin et al. [67] empirically measured that on randomly chosen inputs, the average number of bank conflicts per step is a small constant (between 2 and 3). In contrast, in Chapter 4 we proved the existence of inputs that cause $n/t - o(n/t)$ bank conflicts per step, if n/t and w are coprime. Note that n/t is the trivial upper bound on the number of bank conflicts. These inputs were shown to cause a peak slowdown of $\approx 50\%$ in practice, compared to the runtime on random inputs.

In this chapter, we show that A_i and B_i can be loaded from shared memory into registers, where elements can then be processed internally, without incurring any bank conflicts and with minimal overhead. This approach greatly simplifies the shared memory analysis in balanced two-way divide-and-conquer algorithms, as without bank conflicts, the analysis becomes equivalent to PRAM. We refer this procedure as the *load-balanced dual subsequence gather*.[1] We demonstrate experimentally on the case of mergesort, that the runtime using our bank conflict free load-balanced dual subsequence gather is essentially the same as on random inputs, but holds for all inputs, even the worst case ones. Therefore, eliminating the slowdowns due to bank conflicts in practice.

5.1 Preliminaries

Memory organized into distributed memory modules are not unique to GPUs, emerging as early as the 1980s [53]. Such memory design has been modeled using the *Distributed Memory Machine (DMM)* [76] and the analysis of the delay, due to congestion on memory modules, of an arbitrary PRAM algorithm is known as the *granularity of parallel memories* problem [26, 27, 28, 34, 63, 76, 81, 103]. The DMM can be used to model shared memory accesses on a GPU, because of the natural mapping of shared memory banks to memory modules of the DMM and threads of a warp to DMM processors. For a DMM consisting of w memory modules and w synchronous processors, Czumaj et al. [27] present an access schedule that results in a $O(\log\log\log w \log^* w)$ factor delay, with high probability. In contrast, a naive PRAM implementation can incur a $O(w)$ factor slowdown in the worst case.

Unfortunately in practice, the overheads associated with the techniques used in these general approaches, such as universal hashing, randomization, and data replication, make it impractical for high performance implementations. Alternatively, one can design *bank conflict free* algorithms [1, 36, 20, 51, 66, 68, 82, 112] directly in the DMM model – dedicated algorithms for specific problems that guarantee no bank conflicts. Without any bank conflicts, the runtime analysis becomes much simpler, as it becomes equivalent to PRAM analysis. Compared to standard PRAM approaches, however, bank conflict free algorithms usually come at a cost of increased overhead, e.g., auxiliary memory usage [1, 20, 36, 51], increased code complexity [51, 112], higher constant factors [1, 20], or more overall work [66, 68, 112].

[1]The inverse procedure can be used to write elements from registers into shared memory in a bank conflict free manner, i.e., a load-balanced dual subsequence scatter.

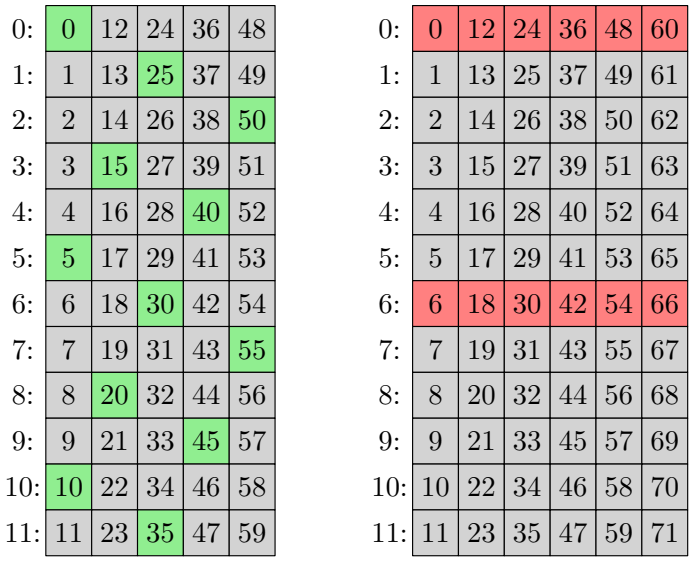

Figure 5.1: Visualization of strided accesses in shared memory with $w = 12$ (number of shared memory banks and number of threads in a warp). The left figure depicts bank conflict free accesses (colored green) resulting from using a coprime stride distance of 5 ($\text{GCD}(5, 12) = 1$). In comparison, the right figure depicts the worst case number of bank conflicts (colored red) resulting from using a not coprime stride distance of 6 ($\text{GCD}(6, 12) = 2$). Cells are numbered with its shared memory index.

The crux of designing bank conflict free algorithms is understanding the mapping of accesses performed to each of the w shared memory banks. It has been observed in previous work [11, 20, 36, 66] that bank conflicts do not occur when accesses by threads of a warp are separated by a distance that is coprime with w (i.e., does not share a common divisor with w). Conversely, bank conflicts do occur when the access stride instead shares a common divisor with w (i.e., not coprime). Figure 5.1 illustrates this behavior on shared memory with $w = 12$, using accesses strided by a distance of 5 (coprime) and 6 (not coprime). Leveraging this observation, researchers have designed bank conflict free algorithms via padding data, staggering accesses, and/or permuting elements into an alternate layout. However, in spite of this common design approach, insight into this behavior has not been fully understood and formalized, leaving researchers to reprove bank conflict free accesses for each problem considered. As number theory is the perfect tool to discover insights into integer mappings, we apply this theoretical foundation to clarify and codify this phenomena. Our analysis in Section 5.2 relies on various number theory results, specifically, we utilize results related to congruences, the greatest common denominator of two integers (see Definition 39), and complete residue systems (see Definition 49). For a full review of relevant number theory results, we refer readers to Appendix A.

Algorithm 2 Reads $|A_i|$ elements from the A list and $|B_i| = E - |A_i|$ elements from the B list from shared memory into registers. Let π be a permutation that reverses the order of elements (described in Section 5.2.1) and ρ performs a circular shift (described in Section 5.2.2).

```
1: LOAD-BALANCED-DUAL-SUBSEQUENCE-GATHER(shmem, items, a_i, b_i, |A_i|, |B_i|)
2:     shmem = ρ(A ∪ π(B))                                  ▷ Permute elements
3:     k = a_i (mod E)
4:     for j = 0 to E − 1
5:         if j − k (mod E) < |A_i|
6:             idx = ρ(j − k (mod E) + a_i)                 ▷ Read (j − k (mod E))-th element of A_i
7:         else
8:             idx = ρ(π(k − j − 1 (mod E) + b_i))          ▷ Read (k − j − 1 (mod E))-th element of B_i
9:         items[j] = shmem[idx]
```

5.2 Load-Balanced Dual Subsequence Gather

The load-balanced dual subsequence gather is a bank conflict free algorithm for loading subsequences from at most two sequences, from shared memory into register space. Intuitively, it is a simple algorithm consisting of a permutation and a scan of A_i in ascending order and B_i in descending order (see Algorithm 2). As mentioned in Section 5.1, we use number theory to prove bank conflict free accesses, namely we construct *complete residue systems* modulo w and use various results related to the greatest common divisor, denoted GCD, of two integers in our proofs.

To explain the basic ideas of our algorithm, in Sections 5.2.1 and 5.2.2, we consider a single warp with its elements from A and B stored in contiguous shared memory locations. We start in Section 5.2.1 by considering values of w and E that are coprime and show that by reversing B and dynamically staggering the scan results in bank conflict free accesses. In Section 5.2.2, we resolve the bank conflicts caused by strided access using a distance E that is not coprime with w by performing an additional circular shift of elements in shared memory. Lastly, in Section 5.2.3 we extend our approach to a full thread block, leading to a practical implementation in practice.

Table 5.1 describes the main parameters used throughout this section. We assume that the number of threads per thread block, u, is a multiple of w, so that there are $\frac{u}{w}$ complete warps in a thread block. We refer to the subsequences of $\mathcal{A}$ and $\mathcal{B}$ for a thread block as A and B; and the offsets of A_i and B_i in A and B as a_i and b_i, respectively.

5.2.1 Coprime

Accessing elements in shared memory with a stride distance that is coprime, relative to the number of banks, has been commonly used to obviate bank conflicts. We start by formalizing this pattern in the context of number theory and show that using a coprime stride distance results in a complete residue system.

Table 5.1: Descriptions of the main parameters for the load-balanced dual subsequence gather.

Parameter	Description
A	Continuous subsequence of $\mathcal{A}$ for a thread block
B	Continuous subsequence of $\mathcal{B}$ for a thread block
A_i	Continuous subsequence of A for the i-th thread
B_i	Continuous subsequence of B for the i-th thread
a_i	Offset of A_i in A
b_i	Offset of B_i in B
u	Number of threads per thread block
w	Number of banks in shared memory and the number of threads per warpy
E	Number of elements per thread (i.e., $\|A_i\| + \|B_i\| = E$)
d	Greatest common divisor of w and E (denoted $\text{GCD}(w, E)$)

Lemma 31. *Let $j \in \mathbb{Z}$. If $d = \text{GCD}(w, E) = 1$, then $R_j = \{j + kE : k \in \mathbb{Z} \text{ and } 0 \leq k < w\}$ is a complete residue system.*

Proof. Since $|R_j| = w$, it suffices to show that for all $r_a, r_b \in R_j$, if $r_a \neq r_b$ then $r_a \not\equiv r_b \pmod{w}$. (By definition, each element in $\mathbb{Z}$ is congruent to some element in $\mathbb{Z}_w = \{0, 1, \ldots, w-1\}$. It follows from $r_a \not\equiv r_b \pmod{w}$ that there exists a valid mapping between R_j and $\mathbb{Z}_w$, and hence, a valid mapping between $\mathbb{Z}$ and R_j.) Assume for the sake of contradiction that $r_a \equiv r_b \pmod{w}$. It follows from Corollary 53, that $r_a \equiv r_b \pmod{w} \implies j + aE \equiv j + bE \pmod{w} \implies a \equiv b \pmod{w} \implies a = b$, since $0 \leq a, b < w$. A contradiction. □

Consider an arbitrary warp with its subsequences of A and B stored in contiguous shared memory locations (A stored first and B stored afterwards). For ease of exposition, we refer to the local index (and offsets) of elements in shared memory belonging to a warp. Our approach performs E rounds of shared memory accesses, where in round $j \in \{0, 1, \ldots, E-1\}$, elements located at index $k \in R_j$ are read into register space. It follows from Proposition 31, that there exists a single element in R_j that is located in each shared memory bank. To ensure bank conflict free access, it remains to be shown that in every round j, each thread has exactly a single element to read.

Without loss of generality, consider the elements in A. Since the elements in R_j are separated by E positions and the number of elements that will be accessed by any single thread in A is at most E (i.e., $|A_i| \leq E$), the elements in R_j that reside in the A list will be read by unique threads. Accounting for both A and B, at most 2 elements will be read by any single thread in each round (see Figure 5.2 for an example). To resolve this conflict between A and B, we reverse the order of the elements in B. Recall that $a_i + b_i = iE$ and $|A_i| + |B_i| = E$. The order of elements in the A list remains unchanged, hence, elements of A_i are read in ascending order in rounds:

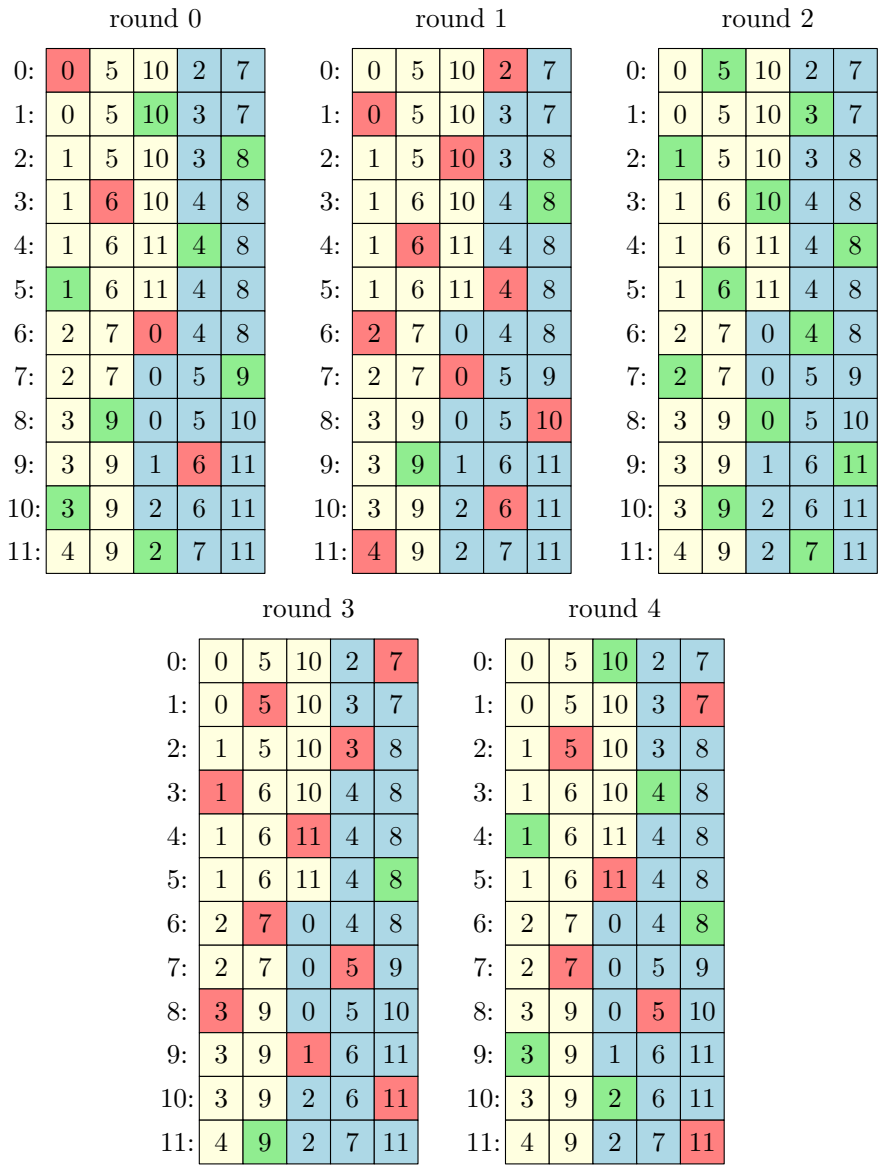

Figure 5.2: Depiction of the read stalls caused by threads in a warp accessing up to 2 elements per round for $w = 12$, $E = 5$, and $d = 1$ (i.e., coprime) on arbitrary input. Cell numbers correspond to the thread that performs the access. Elements colored red cause a stall due to threads needing to access 2 elements in each round.

$$a_i \pmod E, a_i + 1 \pmod E, \ldots, a_i + A_i - 1 \pmod E \,.$$

After reversing the order of elements in the B list, the elements of B_i are now located at indices $\{(wE-1) - b_i, (wE-1) - (b_i+1), \ldots, (wE-1) - (b_i + |B_i| - 1)\}$. Hence, the first element of B_i is read in round $wE - 1 - b_i \equiv a_i - 1 \pmod E$ and the last element of B_i is read in round

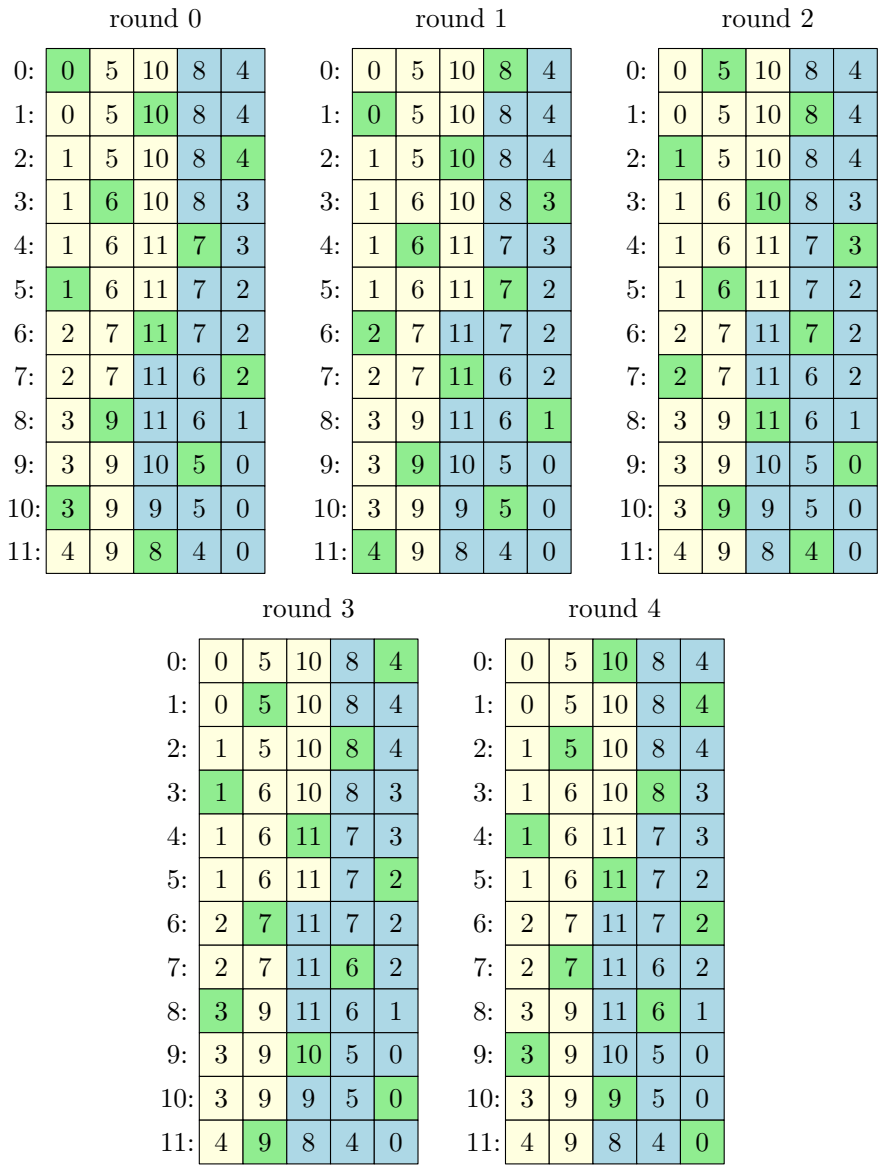

Figure 5.3: Shared memory accesses performed by a warp in the load-balanced dual subsequence gather for $w = 12$, $E = 5$, and $d = 1$ (i.e., coprime) on an arbitrary example input. Elements belonging to the A list (B list) are colored yellow (blue). Cell numbers correspond to the thread that performs the access with cells colored green representing bank conflict free accesses.

$wE - 1 - (b_i + |B_i| - 1) \equiv a_i + A_i \pmod{E}$. Overall, B_i is read in descending order in rounds:

$$a_i + A_i \pmod{E}, a_i + A_i + 1 \pmod{E}, \ldots, a_i - 1 \pmod{E} .$$

Therefore, exactly a single element is read in each round by every thread. Figure 5.3 depicts the accesses performed in each round.

5.2.2 Not Coprime

In Section 5.2.1 (w and E coprime), Lemma 31 shows that for $j \in \mathbb{Z}$, $R_j = \{j + kE : k \in \mathbb{Z} \text{ and } 0 \leq k < w\}$ is a complete residue system modulo w. However, if $d = \text{GCD}(w, E) > 1$ (w and E are not coprime), then $w/d \in \mathbb{Z}$ and every (w/d)-th element in R_j is congruent to each other modulo w. Let $r_a, r_{a+\frac{w}{d}} \in R_j$, $r_a = j + aE \equiv j + aE \pmod{w} \equiv j + \left(a + \frac{w}{d}\right) E = r_{a+\frac{w}{d}}$. Therefore, if E and w are not coprime then R_j is not a complete residue system modulo w. To solve this issue, we partition R_j into d disjoint subsets, each consisting of w/d elements. For $j \in \{0, 1, \ldots, E-1\}$ and $\ell \in \{0, 1, \ldots, d-1\}$, let

$$R_j^{(\ell)} = \left\{ j + \left(\frac{\ell w}{d} + k \right) E : k \in \mathbb{Z} \text{ and } 0 \leq k < \frac{w}{d} \right\}$$

$$\text{and } D_\ell = \left\{ \ell + kd : k \in \mathbb{Z} \text{ and } 0 \leq k < \frac{w}{d} \right\} .$$

We show that the elements in each subset $R_j^{(\ell)}$ are congruent to the elements in $D_{j \pmod d}$. Hence, to construct a complete residue system modulo w, we shift subsets so that each resulting set R'_j contains a single partition that is congruent to a unique D_ℓ.

Lemma 32. *Let $j' \in \{0, 1, \ldots, d-1\}$ such that $j \equiv j' \pmod d$ (i.e., $j = qd + j'$ for some $q \in \mathbb{Z}$). Consider the sets $R_j^{(\ell)}$ and $D_{j'}$.*

1. *For all $r_a \in R_j^{(\ell)}$, there exists $d_b \in D_{j'}$ such that, $r_a \equiv d_b \pmod w$.*
2. *For all $r_a, r_b \in R_j^{(\ell)}$ such that $r_a \neq r_b$, $r_a \not\equiv r_b \pmod w$.*

Proof. Proof of 1: By definition of the greatest common divisor, $\frac{E}{d} \in \mathbb{Z}$. Hence, $r_a = j + (\frac{\ell w}{d} + a)E = j + \frac{\ell w E}{d} + a \cdot \frac{E}{d} \cdot d \equiv j + a \cdot \frac{E}{d} \cdot d \pmod w$. Thus, there exists $a'd \in \mathbb{Z}_w$ such that $0 \leq a' < \frac{w}{d}$ and $a \cdot \frac{E}{d} \equiv a' \pmod w$. If $j < d$, then $r_a \equiv j + a'd \pmod w \equiv d_{a'} \pmod w$. Otherwise, $j \geq d$ and there exists $q \in \mathbb{Z}$ such that $j = qd + j'$, therefore, $r_a \equiv qd + j' + a'd \pmod w \equiv d_{q+a' \pmod{w/d}} \pmod w$.

Proof of 2: We have that $r_a = j + (\frac{\ell w}{d} + a)E \equiv j + a \cdot \frac{E}{d} \cdot d \pmod w \equiv j + a'd \pmod w$ and $r_b = j + (\frac{\ell w}{d} + b)E \equiv j + b \cdot \frac{E}{d} \cdot d \pmod w \equiv j + b'd \pmod w$, for some $a', b' \in \mathbb{Z}_{\frac{w}{d}}$. Therefore, it suffices to show that $a' \not\equiv b' \pmod{w/d}$. Assume for the sake of contradiction, that $a' \equiv b' \pmod{w/d}$. It follows from Corollary 44, that $\text{GCD}(\frac{w}{d}, \frac{E}{d}) = 1$, and hence from Corollary 53, $a' \equiv b' \pmod{w/d} \implies a\left(\frac{E}{d}\right) \equiv b\left(\frac{E}{d}\right) \pmod{w/d} \implies a \equiv b \pmod{w/d} \implies a \not\equiv b \pmod{w/d}$, since $0 \leq a, b < \frac{w}{d}$. A contradiction. □

Observation 33. *$D = \bigcup_{\ell=0}^{d-1} D_\ell$ is a complete residue system modulo w.*

Corollary 34. *Let $R'_j = R_j^{(0)} + R_{j+1 \pmod E}^{(1)} + R_{j+2 \pmod E}^{(2)} + \ldots + R_{j+d-1 \pmod E}^{(d-1)}$. R'_j is a complete residue system modulo w.*

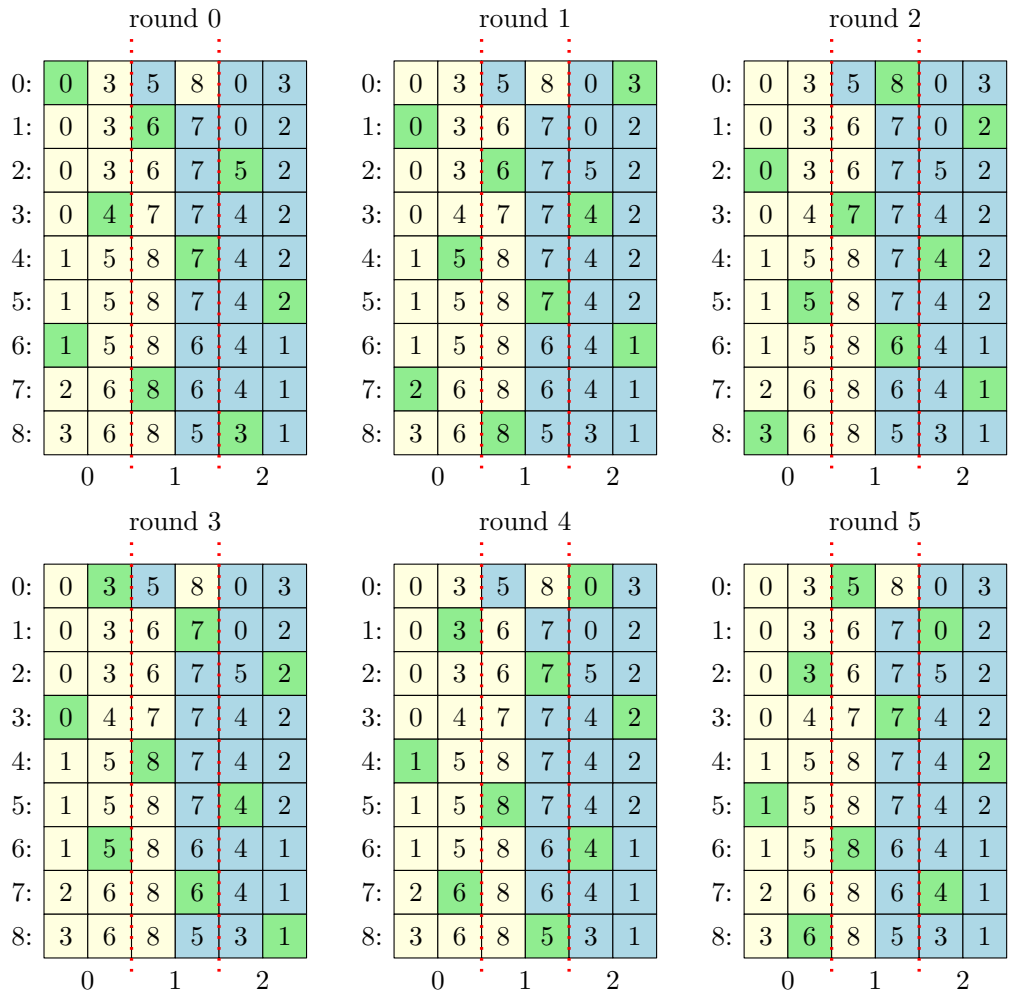

Figure 5.4: Shared memory accesses performed by a warp in the load-balanced dual subsequence gather for $w = 9$, $E = 6$, and $d = 3$ (i.e., not coprime) on an arbitrary example input. Elements belonging to the A list (B list) are colored yellow (blue). The red dotted lines seperate partitions of $wE/d = 16$ elements, that have been circular shifted by 0, 1, and 2 positions, respectively. Cell numbers correspond to the thread that performs the access with cells colored green representing bank conflict free accesses.

Proof. It follows from Lemma 32 that each partition of R'_j is congruent to $D_{j'}$. Since R'_j is the union of consecutive indexed partitions (in a circular manner), each partition is congruent to a unique $D_{j'}$. Therefore, R'_j is a complete residue system modulo w. □

Lemma 35. *Consider the last element in $R_j^{(\ell)}$, denoted a, and the first element in $R_{j+1 \pmod E}^{(\ell+1)}$, denoted b. The difference $(b-a)$ is $(E+1)$, if $j < (E-1)$, and 1 otherwise.*

Proof. Case 1: $j < E-1 \implies b - a = \left(j + 1 + \left(\frac{(\ell+1)w}{d}\right) E\right) - \left(j + \left(\frac{(\ell+1)w}{d} - 1\right) E\right) = E + 1.$

Case 2: $j = E-1 \implies b - a = \left(\left(\frac{(\ell+1)w}{d}\right) E\right) - \left((E-1) + \left(\frac{(\ell+1)w}{d} - 1\right) E\right) = 1.$ □

Lemma 35 shows that for $0 \leq j \leq E - d$, the distance between all elements in R'_j is at least E; and for $E - d < j < E$, the distance between all elements in R'_j is at least E except for a single pair of neighboring elements (i.e., a distance of 1). Ideally, we want the access pattern to match the one used in Section 5.2.1, so that the elements in R'_j are separated by a distance of exactly E. By construction of R'_j, any element indexed at location $k \in \{0, 1, \ldots, wE-1\}$ will be read in round $j \equiv k - \left\lfloor \frac{kd}{wE} \right\rfloor \pmod{E}$. Notice that for $k = \ell \cdot \frac{wE}{d}$, the element indexed at k is read in round $\ell \cdot \frac{wE}{d} - \left\lfloor \ell \cdot \frac{wE}{d} \cdot \frac{d}{wE} \right\rfloor = \ell \cdot \frac{wE}{d} - \ell \equiv -\ell \pmod{E}$. Furthermore for $x \in \{0, 1, \ldots, \frac{wE}{d} - 1\}$, the element at index $(k+x)$ is read in round $x-\ell \pmod{E}$. Intuitively, each partition of $\left(\frac{wE}{d}\right)$ elements has an access pattern that is circular shifted by ℓ rounds relative to the access pattern of the 0-th partition (elements indexed $\{0, 1, \ldots, \frac{wE}{d} - 1\}$). Therefore, we align the accesses to elements in the ℓ-th partition by performing a circular shift of ℓ locations. After shifting elements, any element originally indexed at location k is read in round $j \equiv k \pmod{E}$. As in Section 5.2.1, to resolve read conflicts between lists, we additionally reverse the order of the elements in B. Figure 5.4 illustrates the accesses for values of w and E that are not coprime.

5.2.3 Thread Block

Consider an arbitrary warp $v \in \{0, 1, \ldots, \frac{u}{w} - 1\}$ in the thread block and let α_v and β_v be the index of the first element in the A list and B list for the warp, respectively. Since there are at most vwE elements from the A list assigned to the previous $(v-1)$ warps, $\alpha_v \in \{0, 1, \ldots, vwE - 1\}$. We extend the permutation used in Section 5.2.1 to reverse all elements in the B list for the full thread block. After this reversal, $\beta_v = (uwE - 1) - (vwE - \alpha_v) = (u - v)wE + \alpha_v - 1$. Let $|A_v|, |B_v| \in \mathbb{Z}^+$ be the number of elements in the A and B list assigned to warp v, such that $|A_v| + |B_v| = wE \iff |B_v| = wE - |A_v|$. For warp v, the elements of the A list start in memory bank $\alpha_v \pmod{w}$ and end in memory bank $\alpha_v + |A_v| - 1 \pmod{w}$. And, the elements of the B list end in memory bank $(u-v)wE + \alpha_v - 1 - (|B_v| - 1) \equiv \alpha_v + |A_v| \pmod{w}$ and start in memory bank $(u-v)wE + \alpha_v - 1 \equiv \alpha_v - 1 \pmod{w}$.

Therefore, each warp can view the resulting memory layout as wE elements stored in contiguous memory locations (starting in an arbitrary memory bank). For values of E such that $\text{GCD}(w, E) > 1$ (i.e., not coprime), the permutation ρ is similarly extended where each partition $\ell \in \{0, 1, \ldots, \frac{ud}{w} - 1\}$ of $\left(\frac{wE}{d}\right)$ elements are circular shifted by $\ell \pmod{d}$ positions, with accesses in each partition shifted accordingly. Figure 5.5 illustrates the accesses for a full thread block.

5.3 Experiments

We evaluate the performance of the load-balanced dual subsequence gather by incorporating the algorithm into the implementation of pairwise mergesort provided in Thrust 1.9.9 [88], which we refer to as `CF-Merge`. In our experiments, we compare `CF-Merge` to the unmodified Thrust merge-

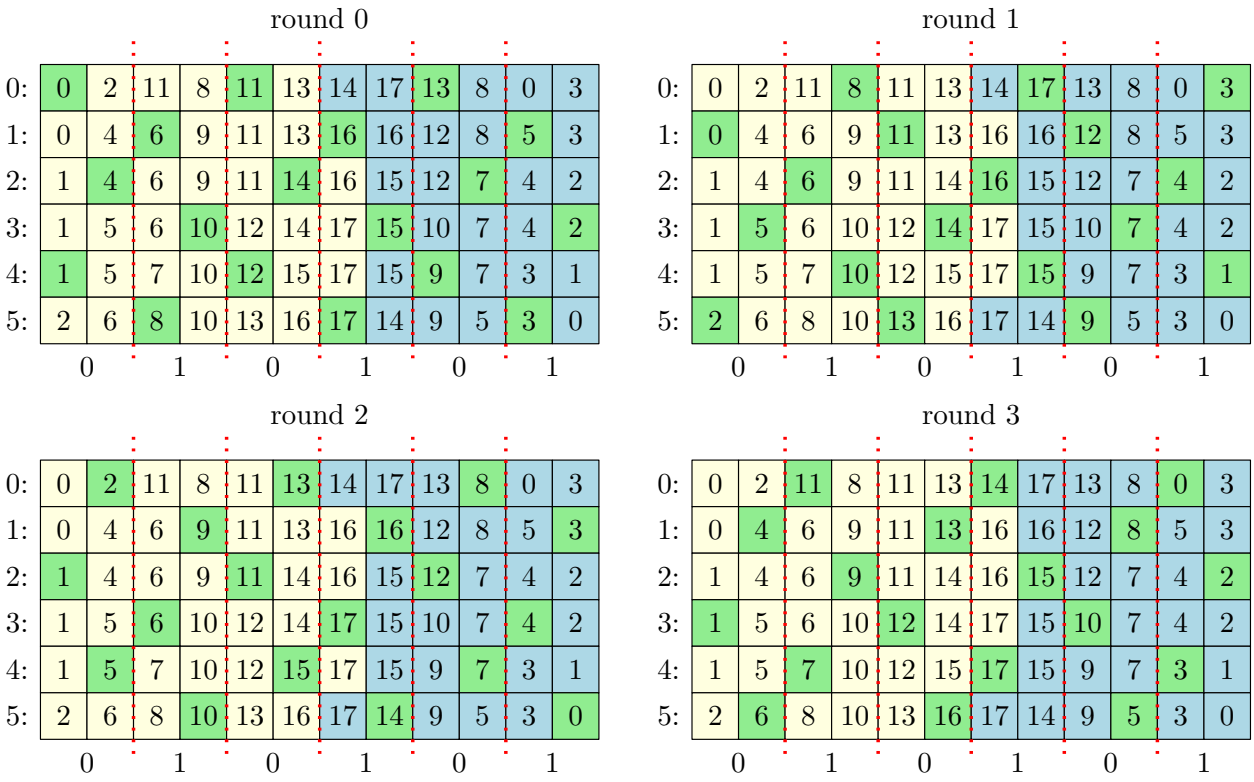

Figure 5.5: Shared memory accesses performed by a thread block in the load-balanced dual subsequence gather for $u = 18$, $w = 6$, $E = 4$, and $d = 2$ (i.e., not coprime) on an arbitrary example input. Elements belonging to the A list (B list) are colored yellow (blue). The red dotted lines seperate partitions of $wE/d = 12$ elements, that have been circular shifted by 0 and 1 positions, respectively. Cell numbers correspond to the thread that performs the access with cells colored green representing bank conflict free accesses. Note that threads in different warps do not cause bank conflicts.

sort implementation on both uniform random inputs and the constructed worst-case inputs from Chapter 4. Recall that in Chapter 4.3, we observed that Thrust uses the software parameters $E = 17$ and $u = 256$, while the parameters $E = 15$ and $u = 512$ provides better performance, on random inputs. This performance difference can be attributed to the corresponding occupancy[2], with $E = 15$ and $u = 512$ providing the optimal 100% theoretical occupancy. Likewise, we compare the performance of these software parameters.

Our implementation of `CF-Merge` uses the thread block approach described in Section 5.2.3. As the permutations performed only rely on information on the total size of each list, each thread block reorders elements during the initial transfer from global memory into shared memory. Furthermore, because both $E = 15$ and $E = 17$ are coprime with $w = 32$, only the coprime variant is implemented. Once elements have been read into register space via the load-balanced dual subsequence gather,

[2]Ratio of active warps to the maximum number of active warps, per SM.

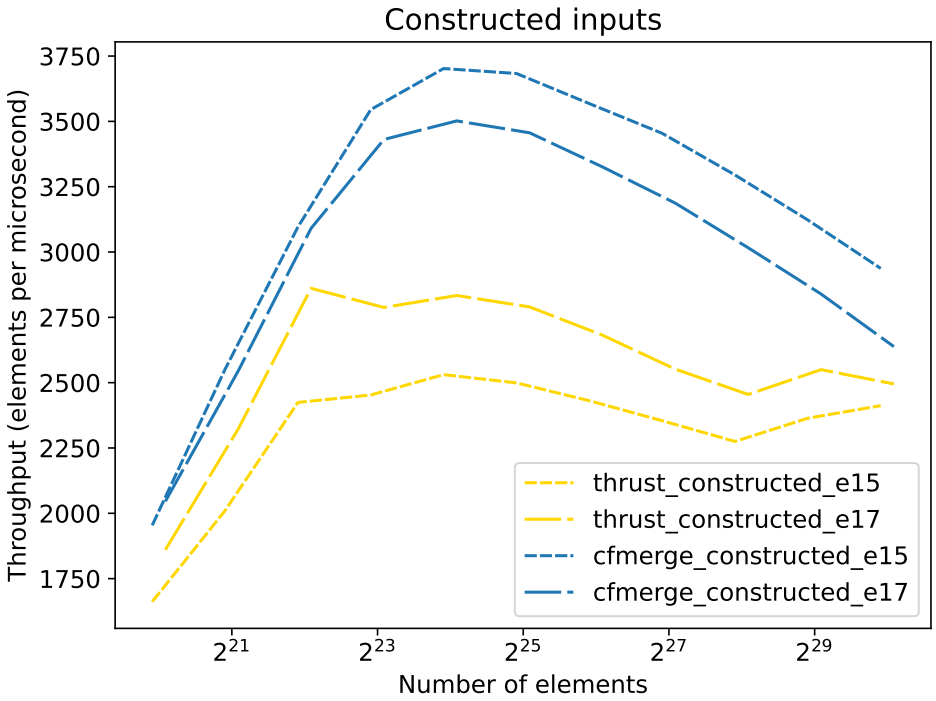

Figure 5.6: Throughput results (elements per microsecond) for Thrust and `CF-Merge` on a NVIDIA RTX 2080 Ti using the constructed worst-case inputs. Thrust results are in yellow and `CF-Merge` results are in blue. The short dashed lines represent software parameters $E = 15$ and $u = 512$; and the long dashed lines represent software parameters $E = 17$ and $u = 256$. The x-axis is displayed on a logarithmic scale.

threads process elements internally. In practice, on NVIDIA GPUs using the CUDA compiler, register memory requires static access as dynamic access to internal data are instead compiled into local memory space. One solution is to use data-oblivious approaches and in our implementation, we adopt the odd-even transposition sort [56] provided in Thrust to process elements in register space.

We conduct experiments using $n = \{2^i E : 16 \leq i \leq 26\}$ 4-byte integers on a NVIDIA RTX 2080 Ti featuring 4,352 total physical cores, 11 GB of global memory, and 96 KiB of unified L1 cache and shared memory (configured to be 32 KiB of L1 cache and 64 KiB of shared memory, or vice versa) per streaming multiprocessor (SM)[3]. All code is written using CUDA 11 [87] and compiled with the `-O3` and `-use_fast_math` optimization flags. Runtimes are recorded via `cudaEventRecord`, with the average across 10 runs being reported. The code used in these experiments can be found at: `https://github.com/algoparc/GPU-CFMerge`.

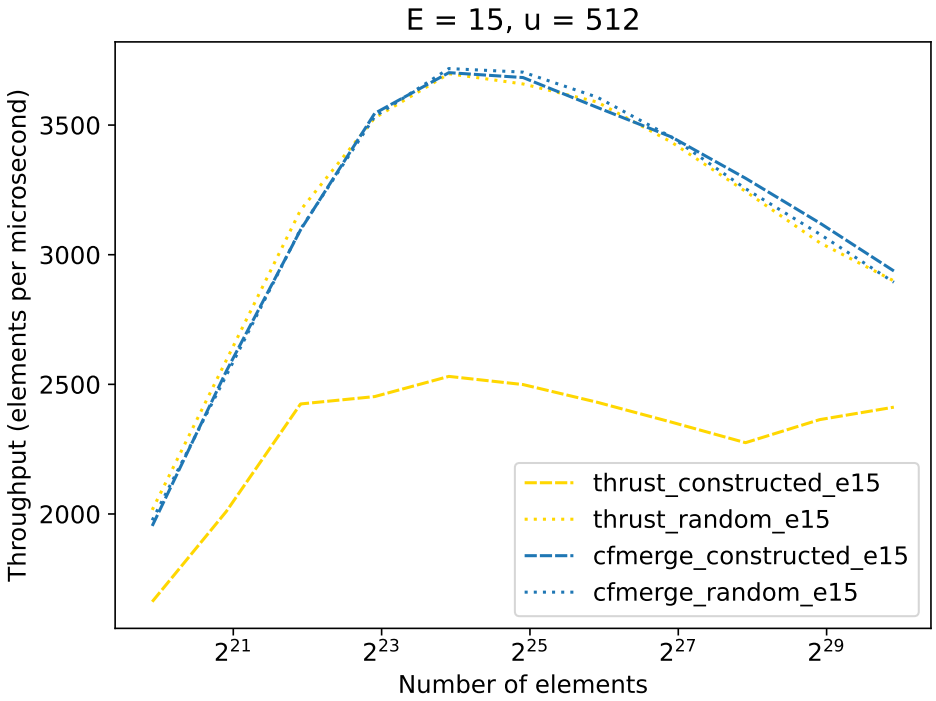

Figure 5.7: Throughput results (elements per microsecond) for Thrust and `CF-Merge` on a NVIDIA RTX 2080 Ti using parameters $E = 15$ and $u = 512$. Thrust results are in yellow and `CF-Merge` results are in blue. The dashed lines represent the constructed worst-case inputs and the dotted lines represent uniform random inputs. The x-axis is displayed on a logarithmic scale.

5.3.1 Results

Figure 5.6 shows the throughput results (elements per microsecond) for both software parameters on the constructed worst-case inputs. Results show that on these inputs, `CF-Merge` provides an average, mean, and maximum speedup of 1.37, 1.45, and 1.47 for $E = 15$ and $u = 512$; and 1.17, 1.23, and 1.25 for $E = 17$ and $u = 256$. This highlights the performance benefits of `CF-Merge`, which uses the bank conflict free load-balanced dual subsequence gather, compared to the unmodified Thrust implementation, which on these inputs incurs the asymptotic worst-case number of bank conflicts. In contrast, on random inputs `CF-Merge` achieves performance comparable to the unmodified Thrust, which has been empirically shown previously to incur a small constant number of bank conflicts (between 2-3) [67]. This illustrates that the runtime overhead associated with performing the load-balanced dual subsequence gather is insignificant in practice. Overall, these results validate that `CF-Merge` obviates the observed slowdown incurred by bank conflicts in shared memory, thereby providing fast runtimes on all possible inputs. Results for both the constructed worst-case inputs and random inputs are shown for each software parameter in Figure 5.7 and Figure 5.8.

[3]GB = 10^9 bytes and KiB = 2^{10} bytes.

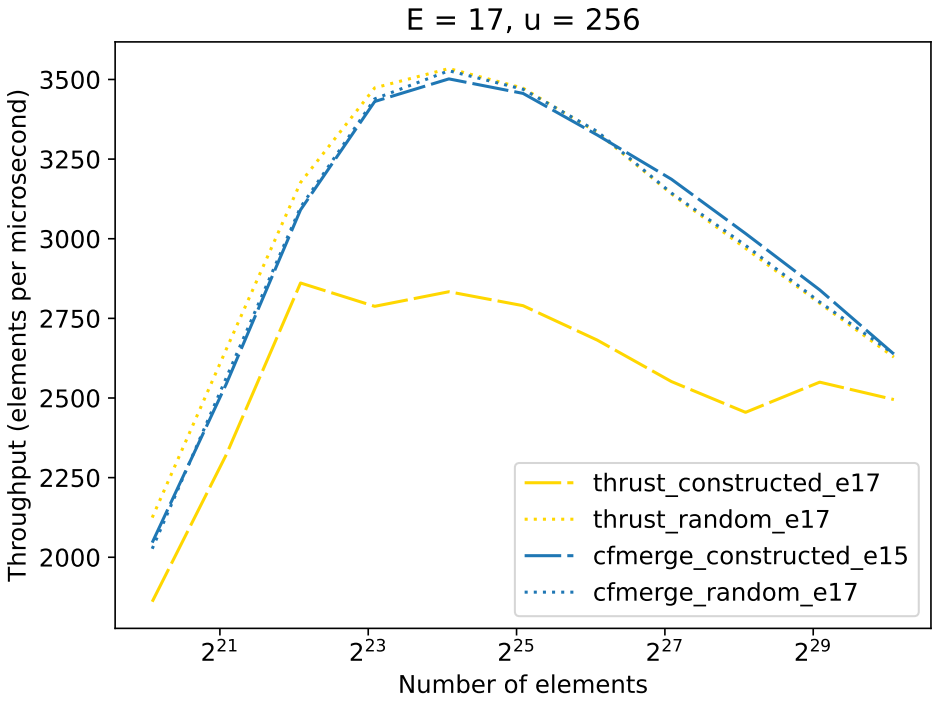

Figure 5.8: Throughput results (elements per microsecond) for Thrust and `CF-Merge` on a NVIDIA RTX 2080 Ti using parameters $E = 17$ and $u = 256$. Thrust results are in yellow and `CF-Merge` results are in blue. The dashed lines represent the constructed worst-case inputs and the dotted lines represent uniform random inputs. The x-axis is displayed on a logarithmic scale.

5.4 Conclusion

In conclusion, this paper addresses the challenges associated with shared memory performance and the analysis of worst-case scenarios caused by bank conflicts in GPU algorithms. Leveraging the Distributed Memory Machine and principles from number theory (e.g., Euclid's Division Lemma, the greatest common divisor, congruences, and complete residue systems), we demonstrate how to design bank conflicts free algorithms on GPUs.

Particularly, we showed that it is possible to eliminate all bank conflicts for a class of algorithms called balanced two-way divide-and-conquer algorithms. Our proposed approach, the load-balanced dual subsequence gather, eliminates bank conflicts with minimal overhead by efficiently loading data from shared memory into registers for processing. This approach greatly simplifies shared memory analysis for these algorithms, as without bank conflicts, shared memory analysis becomes equivalent to PRAM. We validated our approach on GPU mergesort, with experimental results showing that we effectively eliminate the slowdowns caused by bank conflicts in practice.

One avenue of future work is to explore the potential impact of using the load-balanced dual subsequence gather in other balanced two-way divide-and-conquer algorithms. We acknowledge that the effectiveness of our approach relies on the fast processing of elements in registers, which on

GPUs are constrained to static access patterns known at compile time. While our implementation for mergesort utilizes an odd-even sorting network to sort elements in registers, other algorithms may require novel or sophisticated approaches. Further investigation into the capabilities of computation in register space (e.g., oblivious algorithms, circuits, and sorting networks) can yield valuable insights.

Overall, our findings contribute to a deeper understanding of GPU optimization techniques in shared memory and emphasize the importance of considering additional performance metrics beyond parallelism and global memory access. By eliminating bank conflicts, we can further enhance the efficiency of GPU algorithms, leading to faster and more effective high-performance computing on GPUs. This research opens up new possibilities for improving GPU performance and underscores the value of considering shared memory performance and worst-case analysis in the design and implementation of high-performance algorithms on GPUs.

CHAPTER 6
CONCLUSION

GPUs are highly parallel architectures, featuring thousands of physical cores and low latency context switching capabilities, thereby allowing the utilization of hundreds of thousands of threads. However, due to hierarchy of memory units, each with its own latency, bandwidth, and optimal access requirements, designing and analyzing algorithms on GPUs that result in fast GPU implementations, in practice, can be a very challenging task.

In this dissertation, we demonstrated the effective use of various classical parallel models of computation can be used to analyze and design GPU algorithms at various levels of detail, based on the specific problem requirements and research goals. In Chapter 2 and Chapter 3, we examined performance metrics at the highest level of detail on the GPU, the degree of parallelism and the number of accesses to global memory, and utilized the Parallel Random Access Machine (PRAM) model [62] and the Parallel External Memory model [4], respectively. Chapters 4 and 5 focused on lower-level details, specifically shared memory accesses and bank conflicts, and showed how the Distributed Memory Machine (DMM) model [76] combined with various number theory results (e.g., Euclid's Division Lemma, congruences, greatest common divisor, and complete residue systems) can be used to analyze bank conflicts in shared memory and to design bank conflict free algorithms. Importantly, the theoretical design and analysis performed throughout this dissertation were all validated through corresponding experimental results on modern GPUs. By performing both theoretical and experimental research, our work establishes a strong connection between the highlighted parallel models of computation and practical GPU algorithm development.

In summary, this dissertation showcases how classical parallel models of computation can be used for the design and analysis of efficient GPU algorithms. This approach provides valuable insights into the performance of GPU algorithms and guides the identification of optimization opportunities. Moreover, the correspondence between theoretical observations in our selected models and experimental results validates the effectiveness and practical relevance and effectiveness of our approach. By leveraging these parallel models, a deeper understanding of GPU performance factors can be gained, enabling the development of parallel cache-efficient algorithms for GPUs.

Moving forward, it may be worthwhile to explore other models of computation for analyzing additional aspects of the GPU. In particular, investigating parallel models that incorporate synchronization between threads at various levels of the thread hierarchy, as well as models that focus on efficient computation in register space, present promising avenues for future work. By exploring these areas, additional insights can be gained to aid in the design of algorithms that can effectively utilize the GPU architecture, hence, further enhancing the understanding and development of high-performance GPU algorithms.

APPENDIX A
NUMBER THEORY

In this appendix, the proofs for the following results can be found in Andrews [3], with the exception of Corollary 43 and Corollary 44, which we provide.

Lemma 36. *(Euclid's Division Lemma) For any integers a and b such that $b > 0$, there exists unique integers q and r such that $0 \leq r < b$ and*

$$a = qb + r \ .$$

Definition 37. *Let $a, b \in \mathbb{Z}$, such that $b \neq 0$. We say "b* divides *a" or "b* is a divisor of *a" if $\frac{a}{b} \in \mathbb{Z}$, denoted $b \mid a$ (or $b \nmid a$ otherwise).*

Definition 38. *Let $a, b, c \in \mathbb{Z}$, such that $c \neq 0$. If $c \mid a$ and $c \mid b$, then c is a* common divisor *of a and b.*

Definition 39. *Let $a, b \in \mathbb{Z}$ and $d \in \mathbb{Z}^+$, such that both a and b are non-zero (note that one of a or b can be zero). We say d is the* greatest common divisor *of a and b, denoted $\text{GCD}(a, b)$, if*

1. *d is a common divisor of a and b; and*
2. *any integer c that is a common divisor of a and b is also a divisor of d (i.e., $c \mid d$).*

Theorem 40. *Let $a, b \in \mathbb{Z}$, such that both a and b are non-zero (note that one of a or b can be zero). $\text{GCD}(a, b)$ exists and is unique.*

Theorem 41. *Let $a, b, c, d \in \mathbb{Z}$, such that $a \neq 0$, $b \neq 0$, and $d = \text{GCD}(a, b)$. There exists $x, y \in \mathbb{Z}$, such that*

$$ax + by = c$$

if and only if $d \mid c$ (i.e., $\frac{d}{c} \in \mathbb{Z}$).

Definition 42. *Let $a, b \in \mathbb{Z}$, such that $a \neq 0$ and $b \neq 0$. If $\text{GCD}(a, b) = 1$, then a and b are* coprime *(also known as* relatively prime *or* mutually prime*).*

Corollary 43. *Let $a, b \in \mathbb{Z}^+$, such that $a \geq b$. Let $q, r \in \mathbb{Z}$ such that, $a = qb + r$.*

$$\text{GCD}(a, b) = \text{GCD}(b, r) \ .$$

Proof. For ease of notation, let $d = \text{GCD}(a, b)$. By definition, $d \mid a$ and $d \mid b$, hence, there exists positive integers x, y such that $a = dx$ and $b = dy$.

$$r = a - qb$$

$$= dx - qdy$$
$$= d(x - qy) \ .$$

Thus, $d = \text{GCD}(a, b) \mid r$ and $\text{GCD}(a, b) \leq \text{GCD}(b, r)$. We use a similar argument to show that $\text{GCD}(b, r) \mid a$ and $\text{GCD}(b, r) \leq \text{GCD}(a, b)$. Therefore, $\text{GCD}(a, b) = \text{GCD}(b, r)$. □

Corollary 44. *Let $a, b \in \mathbb{Z}$ and $d = \text{GCD}(a, b)$.*

$$\text{GCD}\left(\frac{a}{d}, \frac{b}{d}\right) = 1 \ .$$

Proof. Assume c is a positive common divisor of a and b such that $c \mid \frac{a}{d}$ and $c \mid \frac{b}{d}$. In other words, $\frac{a/d}{c} \in \mathbb{Z}^+$ and $\frac{b/d}{c} \in \mathbb{Z}^+$. Hence, there exists $x, y \in \mathbb{Z}^+$ such that $\frac{a}{d} = cx \implies a = cdx$ and $\frac{b}{d} = cy \implies b = cdy$. Thus, cd is a positive common divisor of a and b. Since d is the greatest common divisor of a and b, c must be equal to 1. Therefore, $c = 1$ is the greatest common divisor of $\frac{a}{d}$ and $\frac{b}{d}$. □

Definition 45. *Let $a, b \in \mathbb{Z}$ and $m \in \mathbb{Z}^+$. We say "a is congruent to b modulo m" or "b is a residue of a modulo m", written*

$$a \equiv b \pmod{m} \ ,$$

if $m \mid (a - b)$ (i.e., there exists $q \in \mathbb{Z}$ such that $a = qm + b$).

Theorem 46. *(Congruences are reflexive, symmetric, and transitive.) Let $a, b, c \in \mathbb{Z}$ and $m \in \mathbb{Z}^+$.*

1. *Reflexive: $a \equiv a \pmod{m}$*
2. *Symmetric: if $a \equiv b \pmod{m}$ then $b \equiv a \pmod{m}$*
3. *Transitive: if $a \equiv b \pmod{m}$ and $b \equiv c \pmod{m}$, then $a \equiv c \pmod{m}$*

Theorem 47. *(Congruences can be correctly added, subtracted, and multiplied.) Let $a, b, a', b' \in \mathbb{Z}$ and $m \in \mathbb{Z}^+$, such that*

$$a \equiv a' \pmod{m}$$
$$\text{and } b \equiv b' \pmod{m} \ .$$

Then

$$a \pm b \equiv a' \pm b' \pmod{m}$$
$$\text{and } ab \equiv a'b' \pmod{m} \ .$$

Theorem 48. *(Cancellation Law.) Let $a, b, c \in \mathbb{Z}$ and $m \in \mathbb{Z}^+$. If*

$$ab \equiv ac \pmod{m}$$

and $\text{GCD}(a, m) = 1$*, then*

$$b \equiv c \pmod{m}.$$

Definition 49. *Let $m \in \mathbb{Z}^+$. The set $R = \{r_0, r_1, \ldots, r_{m-1}\}$ is a* complete residue system *modulo m if the following are satisfied:*

1. *for each $i, j \in \{0, 1, \ldots, m-1\}$ such that $i \neq j$,*

$$r_i \not\equiv r_j \pmod{m}$$

2. *for each $n \in \mathbb{Z}$, there exists $r_i \in R$ such that $n \equiv r_i \pmod{m}$.*

Corollary 50. *Let $m \in \mathbb{Z}^+$. The set $\mathbb{Z}_m = \{0, 1, 2, \ldots, m-1\}$ is a complete residue system.*

Theorem 51. *Let $a, b, d, x \in \mathbb{Z}$ and $m \in \mathbb{Z}^+$, such that $d = \text{GCD}(a, m)$. The equation,*

$$ax \equiv b \pmod{m}$$

has exactly d unique solutions modulo m if d divides b (i.e., $\frac{b}{d} \in \mathbb{Z}$) and does not have any solutions otherwise.

Definition 52. *Let $a, b \in \mathbb{Z}$ and $m \in \mathbb{Z}^+$. If $ab \equiv 1 \pmod{m}$, then b is an* inverse *of a modulo m.*

Corollary 53. *Let $n \in \mathbb{Z}$ and $m \in \mathbb{Z}^+$. If $\text{GCD}(n, m) = 1$, then n has a single unique inverse modulo m.*

BIBLIOGRAPHY

[1] Peyman Afshani and Nodari Sitchinava. Sorting and permuting without bank conflicts on GPUs. In *European Symposium on Algorithms*, volume 9294 of *Lecture Notes in Computer Science*, pages 13–24. Springer, 2015.

[2] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[3] George E. Andrews. *Number Theory*. Dover Publications, Inc., 1994.

[4] Lars Arge, Michael T. Goodrich, Michael J. Nelson, and Nodari Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *Symposium on Parallelism in Algorithms and Architectures*, pages 197–206. ACM, 2008.

[5] Mikhail J. Atallah, Danny Z. Chen, and Ovidiu Daescu. Efficient parallel algorithms for planar *st*-graphs. *Algorithmica*, 35(3):194–215, 2003.

[6] Nikolai Baudis, Florian Jacob, and Philipp Andelfinger. Performance evaluation of priority queues for fine-grained parallel tasks on GPUs. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 1–11. IEEE Computer Society, 2017.

[7] Sean Baxter. Modern GPU. https://github.com/moderngpu/moderngpu, 2016.

[8] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proc. of ACM SIGFIDET Workshop on Data Description, Access and Control*, pages 107–141, 1970.

[9] Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.

[10] Kyle Berney, Henri Casanova, Alyssa Higuchi, Ben Karsin, and Nodari Sitchinava. Beyond binary search: Parallel in-place construction of implicit search tree layouts. In *International Parallel and Distributed Processing Symposium*, pages 1070–1079. IEEE Computer Society, 2018.

[11] Kyle Berney and Nodari Sitchinava. Engineering worst-case inputs for pairwise merge sort on GPUs. In *International Parallel and Distributed Processing Symposium*, pages 1133–1142. IEEE Computer Society, 2020.

[12] Guy E. Blelloch, Yan Gu, Yihan Sun, and Kanat Tangwongsan. Parallel shortest paths using radius stepping. In *Symposium on Parallelism in Algorithms and Architectures*, pages 443–454. ACM, 2016.

[13] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, 1974.

[14] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. Cache oblivious search trees via binary trees of small height. In *Symposium on Discrete Algorithms*, pages 39–48. ACM/SIAM, 2002.

[15] Gerth Stølting Brodal, Rolf Fagerberg, Ulrich Meyer, and Norbert Zeh. Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths. In *Scandinavian Workshop on Algorithm Theory*, volume 3111 of *Lecture Notes in Computer Science*, pages 480–492. Springer, 2004.

[16] Gerth Stølting Brodal, Jesper Larsson Träff, and Christos D. Zaroliagis. A parallel priority queue with constant time operations. *J. Parallel Distributed Comput.*, 49(1):4–21, 1998.

[17] Federico Busato and Nicola Bombieri. An efficient implementation of the bellman-ford algorithm for kepler GPU architectures. *IEEE Trans. Parallel Distributed Syst.*, 27(8):2222–2233, 2016.

[18] Nairen Cao, Jeremy T. Fineman, and Katina Russell. Parallel shortest paths with negative edge weights. In *Symposium on Parallelism in Algorithms and Architectures*, pages 177–190. ACM, 2022.

[19] Thomas C. Carroll and Prudence W. H. Wong. An improved abstract GPU model with data transfer. In *International Conference on Parallel Processing Workshops*, pages 113–120. IEEE Computer Society, 2017.

[20] Bryan Catanzaro, Alexander Keller, and Michael Garland. A decomposition for in-place matrix transposition. In *Symposium on Principles and Practice of Parallel Programming*, pages 193–206. ACM, 2014.

[21] Shiva Chaudhuri and Christos D. Zaroliagis. Shortest paths in digraphs of small treewdith. part II: optimal parallel algorithms. *Theor. Comput. Sci.*, 203(2):205–223, 1998.

[22] Rezaul Alam Chowdhury and Vijaya Ramachandran. Cache-oblivious shortest paths in graphs using buffer heap. In *Symposium on Parallelism in Algorithms and Architectures*, pages 245–254. ACM, 2004.

[23] Edith Cohen. Polylog-time and near-linear work approximation scheme for undirected shortest paths. *J. ACM*, 47(1):132–166, 2000.

[24] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[25] Andreas Crauser, Kurt Mehlhorn, Ulrich Meyer, and Peter Sanders. A parallelization of Dijkstra's shortest path algorithm. In *International Symposium on Mathematical Foundations of Computer Science*, volume 1450 of *Lecture Notes in Computer Science*, pages 722–731. Springer, 1998.

[26] Artur Czumaj, Friedhelm Meyer auf der Heide, and Volker Stemann. Improved optimal shared memory simulations, and the power of reconfiguration. In *Israel Symposium on Theory of Computing and Systems*, pages 11–19. IEEE Computer Society, 1995.

[27] Artur Czumaj, Friedhelm Meyer auf der Heide, and Volker Stemann. Shared memory simulations with triple-logarithmic delay. In *European Symposium on Algorithms*, volume 979 of *Lecture Notes in Computer Science*, pages 46–59. Springer, 1995.

[28] Artur Czumaj, Friedhelm Meyer auf der Heide, and Volker Stemann. Contention resolution in hashing based shared memory simulations. *SIAM Journal on Computing*, 29(5):1703–1739, 2000.

[29] Mehmet Emin Dalkiliç, Elif Acar, and Gorkem Tokatli. A simple shuffle-based stable in-place merge algorithm. In *World Conference on Information Technology*, volume 3 of *Procedia Computer Science*, pages 1049–1054. Elsevier, 2011.

[30] Daniele D'Angeli and Alfredo Donno. Shuffling matrices, kronecker product and discrete fourier transform. *Discrete Appl. Math.*, 233:1–18, 2017.

[31] Andrew A. Davidson, Sean Baxter, Michael Garland, and John D. Owens. Work-efficient parallel GPU methods for single-source shortest paths. In *International Parallel and Distributed Processing Symposium*, pages 349–359. IEEE Computer Society, 2014.

[32] Marc Davio. Kronecker products and shuffle algebra. *IEEE Transactions on Computers*, C-30(2):116–125, 1981.

[33] Persi Diaconis, R.L. Graham, and William M Kantor. The mathematics of perfect shuffles. *Advances in Applied Mathematics*, 4(2):175–196, 1983.

[34] Paul F. Dietz and Rajeev Raman. Very fast optimal parallel algorithms for heap construction. In *Symposium on Parallel and Distributed Processing*, pages 514–521. IEEE Computer Society, 1994.

[35] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[36] Yuri Dotsenko, Naga K. Govindaraju, Peter-Pike J. Sloan, Charles Boyd, and John Manferdelli. Fast scan algorithms on graphics processors. In *International Conference on Supercomputing*, pages 205–213, 2008.

[37] Michael Elkin and Ofer Neiman. Hopsets with constant hopbound, and applications to approximate shortest paths. In *Symposium on Foundations of Computer Science*, pages 128–137. IEEE Computer Society, 2016.

[38] Michael Elkin and Ofer Neiman. Linear-size hopsets with small hopbound, and constant-hopbound hopsets in RNC. In *Symposium on Parallelism in Algorithms and Architectures*, pages 333–341. ACM, 2019.

[39] John Ellis, Tobias Krahn, and Hongbing Fan. Computing the cycles in the perfect shuffle permutation. *Information Processing Letters*, 75(5):217 – 224, 2000.

[40] John A. Ellis, Hongbing Fan, and Jeffrey O. Shallit. The cycles of the multiway perfect shuffle permutation. *Discrete Mathematics & Theoretical Computer Science*, 5(1):169–180, 2002.

[41] John A. Ellis and Minko Markov. In situ, stable merging by way of the perfect shuffle. *The Computer Journal*, 43(1):40–53, 2000.

[42] John A. Ellis and Ulrike Stege. A provably, linear time, in-place and stable merge algorithm via the perfect shuffle. *CoRR*, abs/1508.00292, 2015.

[43] Faith E. Fich, J. Ian Munro, and Patricio V. Poblete. Permuting in place. *SIAM Journal on Computing*, 24(2):266–278, 1995.

[44] Lester R Ford Jr. Network flow theory. Technical report, Rand Corp, Santa Monica, Ca., 1956.

[45] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Symposium on Theory of Computing*, pages 114–118. ACM, 1978.

[46] James Fox, Oded Green, Kasimir Gabert, Xiaojing An, and David A. Bader. Fast and adaptive list intersections on the GPU. In *High Performance Extreme Computing Conference*, pages 1–7. IEEE, 2018.

[47] Michael L. Fredman, Robert Sedgewick, Daniel Dominic Sleator, and Robert Endre Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

[48] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. In *Symposium on Foundations of Computer Science*, pages 338–346. IEEE Computer Society, 1984.

[49] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Symposium on Foundations of Computer Science*, pages 285–298. IEEE Computer Society, 1999.

[50] Isaac Gelado and Michael Garland. Throughput-oriented GPU memory allocation. In *Symposium on Principles and Practice of Parallel Programming*, pages 27–37. ACM, 2019.

[51] Dominik Göddeke and Robert Strzodka. Cyclic reduction tridiagonal solvers on GPUs applied to mixed-precision multigrid. *IEEE Trans. Parallel Distributed Syst.*, 22(1):22–32, 2011.

[52] Andrew V. Goldberg. Scaling algorithms for the shortest paths problem. *SIAM Journal on Computing*, 24(3):494–504, 1995.

[53] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. The NYU Ultracomputer - designing an MIMD shared memory parallel computer. *IEEE Trans. Computers*, 32(2):175–189, 1983.

[54] Chunyang Gou and Georgi Gaydadjiev. Addressing GPU on-chip shared memory bank conflicts using elastic pipeline. *International Journal of Parallel Programming*, 41(3):400–429, 2013.

[55] Oded Green, Robert McColl, and David A. Bader. GPU merge path: a GPU merging algorithm. In *International Conference on Supercomputing*, pages 331–340. ACM, 2012.

[56] A. Nico Habermann. Parallel neighbor-sort (or the glory of the induction principle). Technical Report AD-759 248, Carnegie Mellon University, 1972.

[57] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *High Performance Computing*, volume 4873 of *Lecture Notes in Computer Science*, pages 197–208. Springer, 2007.

[58] Xi He, Dinesh Agarwal, and Sushil K. Prasad. Design and implementation of a parallel priority queue on many-core architectures. In *High Performance Computing*, pages 1–10. IEEE Computer Society, 2012.

[59] Lorenz Hübschle-Schneider and Peter Sanders. Communication efficient algorithms for top-k selection problems. In *International Parallel and Distributed Processing Symposium*, pages 659–668. IEEE Computer Society, 2016.

[60] Galen C. Hunt, Maged M. Michael, Srinivasan Parthasarathy, and Michael L. Scott. An efficient algorithm for concurrent priority queue heaps. *Inf. Process. Lett.*, 60(3):151–157, 1996.

[61] Peiyush Jain. A simple in-place algorithm for in-shuffle. *CoRR*, abs/0805.1598, 2008.

[62] Joseph F. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[63] Richard M. Karp, Michael Luby, and Friedhelm Meyer auf der Heide. Efficient PRAM simulation on a distributed memory machine. In *Symposium on Theory of Computing*, pages 318–326. ACM, 1992.

[64] Richard M. Karp and Vijaya Ramachandran. Handbook of theoretical computer science (vol. a). chapter Parallel Algorithms for Shared-memory Machines, pages 869–941. 1990.

[65] Ben Karsin. *A Performance Model for GPU Architectures: Analysis and Design of Fundamental Algorithms*. PhD thesis, University of Hawaii at Manoa, 2018.

[66] Ben Karsin, Henri Casanova, and Nodari Sitchinava. Efficient batched predecessor search in shared memory on GPUs. In *High Performance Computing*, pages 335–344. IEEE Computer Society, 2015.

[67] Ben Karsin, Volker Weichert, Henri Casanova, John Iacono, and Nodari Sitchinava. Analysis-driven engineering of comparison-based sorting algorithms on GPUs. In *International Conference on Supercomputing*, pages 86–95. ACM, 2018.

[68] Akihiko Kasagi, Koji Nakano, and Yasuaki Ito. An implementation of conflict-free offline permutation on the GPU. In *International Conference on Networking and Computing*, pages 226–232. IEEE Computer Society, 2012.

[69] Paul-Virak Khuong and Pat Morin. Array layouts for comparison-based searching. *J. Exp. Algorithmics*, 22:1.3:1–1.3:39, 2017.

[70] Philip N. Klein and Sairam Subramanian. A linear-processor polylog-time algorithm for shortest paths in planar graphs. In *Foundations of Computer Science*, pages 259–270. IEEE Computer Society, 1993.

[71] Donald Ervin Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1998.

[72] Atsushi Koike and Kunihiko Sadakane. A novel computational model for GPUs with applications to efficient algorithms. *Int. J. Netw. Comput.*, 5(1):26–60, 2015.

[73] Nikolaj Leischner, Vitaly Osipov, and Peter Sanders. GPU sample sort. In *International Symposium on Parallel and Distributed Processing*, pages 1–10. IEEE, 2010.

[74] Shengren Li and Nina Amenta. Brute-force k-nearest neighbors search on the GPU. In *Similarity Search and Applications*, volume 9371 of *Lecture Notes in Computer Science*, pages 259–270. Springer, 2015.

[75] Lin Ma, Kunal Agrawal, and Roger D. Chamberlain. A memory access model for highly-threaded many-core architectures. *Future Gener. Comput. Syst.*, 30:202–215, 2014.

[76] Kurt Mehlhorn and Uzi Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Inf.*, 21:339–374, 1984.

[77] Duane Merrill and Michael Garland. Merge-based parallel sparse matrix-vector multiplication. In John West and Cherri M. Pancake, editors, *High Performance Computing, Networking, Storage and Analysis*, pages 678–689. IEEE Computer Society, 2016.

[78] Duane Merrill and Michael Garland. Merge-based sparse matrix-vector multiplication (SpMV) using the CSR storage format. In *Principles and Practice of Parallel Programming*, pages 43:1–43:2. ACM, 2016.

[79] Bruce Merry. A performance comparison of sort and scan libraries for GPUs. *Parallel Processing Letters*, 25(4):1550007:1–1550007:8, 2015.

[80] Ulrich Meyer and Peter Sanders. Delta-stepping: A parallel single source shortest path algorithm. In *European Symposium on Algorithms*, volume 1461 of *Lecture Notes in Computer Science*, pages 393–404. Springer, 1998.

[81] Friedhelm Meyer auf der Heide, Christian Scheideler, and Volker Stemann. Exploiting storage redundancy to speed up randomized shared memory simulations. *Theor. Comput. Sci.*, 162(2):245–281, 1996.

[82] Koji Nakano. Simple memory machine models for GPUs. In *International Parallel and Distributed Processing Symposium, Workshops and PhD Forum*, pages 794–803. IEEE Computer Society, 2012.

[83] Koji Nakano. The hierarchical memory machine model for GPUs. In *International Symposium on Parallel and Distributed Processing, Workshops and PhD Forum*, pages 591–600. IEEE Computer Society, 2013.

[84] NVIDIA. CUDA toolkit documentation v10.1. https://docs.nvidia.com/cuda/archive/10.1/, 2019.

[85] NVIDIA. CUDA C best practices guide. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html, 2022.

[86] NVIDIA. CUDA C programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2022.

[87] NVIDIA. CUDA toolkit documentation v11. https://docs.nvidia.com/cuda/index.html, 2022.

[88] NVIDIA. Thrust. https://github.com/thrust/thrust, 2022.

[89] NVIDIA and Duane Merrill. CUB. https://nvlabs.github.io/cub/, 2022.

[90] Hagen Peters, Ole Schulz-Hildebrandt, and Norbert Luttenberger. Fast in-place, comparison-based sorting with CUDA: a study with bitonic sort. *Concurrency and Computation: Practice and Experience*, 23(7):681–693, 2011.

[91] Hagen Peters, Ole Schulz-Hildebrandt, and Norbert Luttenberger. A novel sorting algorithm for many-core architectures based on adaptive bitonic sort. In *International Parallel and Distributed Processing Symposium*, pages 227–237. IEEE Computer Society, 2012.

[92] Harald Prokop. Cache-oblivious algorithms. Master's thesis, MIT, 1999.

[93] Christian Ronse. A generalization of the perfect shuffle. *Discret. Math.*, 47:293–306, 1983.

[94] Ran Rui and Yi-Cheng Tu. Fast equi-join algorithms on GPUs: Design and implementation. In *Scientific and Statistical Database Management*, pages 17:1–17:12. ACM, 2017.

[95] Peter Sanders. Randomized priority queues for fast parallel access. *J. Parallel Distributed Comput.*, 49(1):86–97, 1998.

[96] Nadathur Satish, Mark J. Harris, and Michael Garland. Designing efficient sorting algorithms for manycore GPUs. In *International Symposium on Parallel and Distributed Processing*, pages 1–10. IEEE Computer Society, 2009.

[97] Hanmao Shi and Thomas H. Spencer. Time-work tradeoffs of the single-source shortest paths problem. *J. Algorithms*, 30(1):19–32, 1999.

[98] Nodari Sitchinava and Volker Weichert. Provably efficient GPU algorithms. *CoRR*, abs/1306.5076, 2013.

[99] Harold S. Stone. Parallel processing with the perfect shuffle. *IEEE Trans. Computers*, C-20(2):153–161, 1971.

[100] Sairam Subramanian, Roberto Tamassia, and Jeffrey Scott Vitter. An efficient parallel algorithm for shortest paths in planar layered digraphs. *Algorithmica*, 14(4):322–339, 1995.

[101] Sayyada Fahmeeda Sultana and D. Shubhangi. Video encryption algorithm and key management using perfect shuffle. *International Journal of Engineering Research and Applications*, 07:01–05, 2017.

[102] Jesper Larsson Träff and Christos D. Zaroliagis. A simple parallel algorithm for the single-source shortest path problem on planar digraphs. In *Parallel Algorithms for Irregularly Structured Problems*, volume 1117 of *Lecture Notes in Computer Science*, pages 183–194. Springer, 1996.

[103] Eli Upfal and Avi Wigderson. How to share memory in a distributed system. *J. ACM*, 34(1):116–127, 1987.

[104] Jeffrey Scott Vitter. Algorithms and data structures for external memory. *Found. Trends Theor. Comput. Sci.*, 2(4):305–474, 2008.

[105] Kai Wang, Don Fussell, and Calvin Lin. A fast work-efficient SSSP algorithm for GPUs. In *Principles and Practice of Parallel Programming*, pages 133–146. ACM, 2021.

[106] Leyuan Wang, Sean Baxter, and John D. Owens. Fast parallel suffix array on the GPU. In *Euro-Par*, volume 9233 of *Lecture Notes in Computer Science*, pages 573–587. Springer, 2015.

[107] Yangzihao Wang, Andrew A. Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: a high-performance graph processing library on the GPU. In *Principles and Practice of Parallel Programming*, pages 11:1–11:12. ACM, 2016.

[108] Shucai Xiao and Wu-chun Feng. Inter-block GPU communication via fast barrier synchronization. In *International Symposium on Parallel and Distributed Processing*, pages 1–12. IEEE Computer Society, 2010.

[109] Yunlong Xu, Lan Gao, Rui Wang, Zhongzhi Luan, Weiguo Wu, and Depei Qian. Lock-based synchronization for GPU architectures. In *Computing Frontiers*, pages 205–213. ACM, 2016.

[110] Carl Yang, Aydin Buluç, and John D. Owens. Design principles for sparse matrix multiplication on the GPU. In *Euro-Par*, volume 11014 of *Lecture Notes in Computer Science*, pages 672–687. Springer, 2018.

[111] Qingxuan Yang, John A. Ellis, Khalegh Mamakani, and Frank Ruskey. In-place permuting and perfect shuffling using involutions. *Inf. Process. Lett.*, 113(10-11):386–391, 2013.

[112] Yao Zhang, Jonathan Cohen, and John D. Owens. Fast tridiagonal solvers on the GPU. In R. Govindarajan, David A. Padua, and Mary W. Hall, editors, *Symposium on Principles and Practice of Parallel Programming*, pages 127–136. ACM, 2010.