ABSTRACT

Title of dissertation: OPTIMIZATION TECHNIQUES FOR MAPPING ALGORITHMS AND APPLICATIONS ONTO CUDA GPU PLATFORMS AND CPU-GPU HETEROGENEOUS PLATFORMS

Jing Wu, Doctor of Philosophy, 2014

Dissertation directed by: Professor Joseph F JaJa, Department of Electrical and Computer Engineering

An emerging trend in processor architecture seems to indicate the doubling of the number of cores per chip every two years with same or decreased clock speed. Of particular interest to this thesis is the class of many-core processors, which are becoming more attractive due to their high performance, low cost, and low power consumption. The main goal of this dissertation is to develop optimization techniques for mapping algorithms and applications onto CUDA GPUs and CPU-GPU heterogeneous platforms.

The Fast Fourier transform (FFT) constitutes a fundamental tool in computational science and engineering, and hence a GPU-optimized implementation is of paramount importance. We first study the mapping of the 3D FFT onto the recent, CUDA GPUs and develop a new approach that minimizes the number of global memory accesses and overlaps the computations along the different dimensions. We obtain some of the fastest known implementations for the computation of multi-dimensional FFT.

We then present a highly multithreaded FFT-based direct Poisson solver that is optimized for the recent NVIDIA GPUs. In addition to the massive multithreading, our al-

gorithm carefully manages the multiple layers of the memory hierarchy so that all global memory accesses are coalesced into 128-bytes device memory transactions. As a result, we have achieved up to 375GFLOPS with a bandwidth of 120GB/s on the GTX 480.

We further extend our methodology to deal with CPU-GPU based heterogeneous platforms for the case when the input is too large to fit on the GPU global memory. We develop optimization techniques for memory-bound, and computation-bound application. The main challenge here is to minimize data transfer between the CPU memory and the device memory and to overlap as much as possible these transfers with kernel execution. For memory-bounded applications, we achieve a near-peak effective PCIe bus bandwidth, 9-10GB/s and performance as high as 145 GFLOPS for multi-dimensional FFT computations and for solving the Poisson equation. We extend our CPU-GPU based software pipeline to a computation-bound application-DGEMM, and achieve the illusion of a memory of the CPU memory size and a computation throughput similar to a pure GPU.

OPTIMIZATION TECHNIQUES FOR MAPPING ALGORITHMS AND APPLICATIONS ONTO CUDA GPU PLATFORMS AND CPU-GPU HETEROGENEOUS PLATFORMS

by

Jing Wu

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2014

Advisory Committee:
Professor Joseph F JaJa Chair/Advisor
Professor Shuvra S. Bhattacharyya
Professor Tudor Dumitras
Professor Manoj Franklin
Professor Amitabh Varshney

Dedication

To My Grandma

Acknowledgments

I owe my greatest gratitude to my advisor, Dr. Joseph F. JaJa, for being my academic advisor and life mentor during my graduate study at University of Maryland. His insightful opinions and invaluable advice have guided my through many difficult times in research and helped me become a much more knowledgable, confident and mentally stronger version of myself. His positive attitude toward life and professional endeavors would benefit me for my entire life.

I would also like to thank my dissertation examining committee members, Dr. Shuvra S. Bhattacharyya, Dr. Tudor Dumitras, Dr. Manoj Franklin, Dr. Gang Qu, and Dr. Amitabh Varshney, for their precious time and effort to serve on the committee, insightful questions, and helpful comments to improve the quality of this dissertation.

I'm very grateful for having those wonderful group members. Dr. Sangchul Song gave me plenty of help and useful advice during my very first research project in my second year. Dr. Zheng Wei, a long time friend that I met since my day one on campus and a wonderful lab-mate, gave me help in various aspects, both in research and life. I would also like to thank our dedicated lab managers/programmers, Mike Smorul and Michael Ritter, and those wonderful UMIACS help desk staff, Fritz McCall, Derek Yarnell, Jonathan Lent, Shawn Bobbin, and others, for their patient software installation and reinstallation, machine configuration and re-configuration, and timely trouble-shooting, even at midnight and/or on weekends. I would also like to thank my other lab mates: Qi Wang, Wenshuai Hou and Nuttiiya Seekhao for the valuable technical discussions and joyful chats we've enjoyed. In addition, I'm so thankful for the consistent

support from my beloved friends - Dr. Junjun Gu, Dr. Beiyu Rong, Kejia Wang and Jonathan Lent.

My additional thanks go to two authors of some references that I consulted to: Dr. Matteo Frigo and Vasily Volkov for their insights and help about FFTW and CUDA FFT & GEMM respectively.

In the end, I give my special thanks to my sister and my parents, to whom words can't express my appreciation and their love to me enough.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

BC	Boundary Condition
BW	Bandwidth
DGEMM	Double Precision GEneral Matrix Multiplication
APU	Accelerated Processing Unit
CPU	Central Processing Unit
GPU	Graphic Processing Unit
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
FFT	Fast Fourier Transform
DFT	Discrete Fourier Transform
MKL	Math Kernel Library
CUDA	Compute Unified Device Architecture
ALU	Arithmetic Logic Unit
FPU	Floating-Point Unit
SISD	Single Instruction Single Data
SIMD	Single Instruction Multiple Data
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
SSE	Streaming SIMD Extensions
AVX	Advanced Vector Extension
SMT	Simultaneous Multithreading
HT	Hyper-threading
UMA	Uniform Memory Access
NUMA	Non-Uniform Memory Access
OpenMP	Open MultiProcessing
MPI	Message Passing Interface
POSIX	Portable Operating System Interface
RDMA	Remote Direct Memory Access
HPC	High Performance Computing
PVM	Parallel Virtual Machine
FCS	Fiber Channel Standard
SCI	Scalable Coherent Interface
DIT	Decimation in Time
DIF	Decimation in Frequency
DNS	Direct Numerical Simulation
LES	Large-eddy simulations
IB	Immersed-boundary
SM	Streaming Multiprocessor
SP	Streaming Processor
OpenCL	Open Computing Language
PRAM	Parallel Random-Access Machine
OpenACC	Open Accelerators

Chapter 1: Introduction

The advances in computing hardware over the past five decades have followed Moore's Law [2], which states that the density of transistors on a chip doubles approximately every 2 years. At the same time, software has been enjoying performance improvements due to increased clock frequencies enabled by the shrinking size of the transistors. However, the free performance lunch [3] of the general-purpose single-core processors has stopped around 2005, due to a number of physical constraints such as power consumption and wire delays. Parallelism, which comes in various forms, is the only direction forward to enable continued performance improvement for demanding applications. At the lowest level, instruction level, pipelining, superscalar execution, and SIMD vector instruction execution have been widely used in modern processors. At the chip scale, multi-core and many-core processors dominate today. Clusters of multi-socket, multi-core CPU processors are typical in high end computing systems. General purpose graphics processing units, or GPGPUs, have gained a major place in the area of high performance computing (HPC) due to their advantages in performance/cost ratio, energy consumption and improved programmability. GPU-based large scale clusters are promising platforms to address complex problems including those involving big data. Novel algorithms that are aware of the underlying architectures are often required to exploit the

substantial available resources in current HPC platforms. In this dissertation, we consider two types of platforms: single NVIDIA GPU platforms and CPU-GPU heterogeneous platforms, which are the basic components of the GPU based clusters and develop novel methodologies to map complex scientific applications onto such platforms.

1.1 Parallel Computing Architecture

According to Hennessy and Patterson, computer architecture comprises three aspects: instruction set architecture, organization or micro architecture design, and hardware [4]. In this thesis, we take the programmer's perspective in that the architecture of a computing system defines the available computing resources, their interactions, and how these resources can be scheduled and coordinated.

High performance depends heavily on parallelism and exists in several forms. Examples of parallelism [5] in microprocessor design include: *bit level parallelism*, *pipelined instruction execution*, *multiple functional units* and *multiple cores*.

Flynn's taxonomy [6] has been the classic terminology used to distinguish parallel computing architectures based on the concurrent instructions and data streams. It includes:

- ***SISD - Single instruction stream single data stream***

  This is the traditional sequential CPU architecture; at any one time, only a single instruction is executed, operating on a single data time. However, it can exploit instruction-level parallelism, such as pipelining, superscalar and speculative execution.

- ***SIMD - Single instruction stream, multiple data streams***

  For this type, there can be multiple processing units, each operating on its own data item, but they all are executing the same instruction. The SIMD architecture exploits data-level parallelism and includes classic examples such as vector architectures and multimedia extensions to standard instruction sets.

- ***MIMD - Multiple instruction stream, multiple data streams***

  Here, multiple processors operate on multiple data items, each owning its own memory and executing its own independent instructions. The MIMD architecture is more flexible than SIMD and is more generally applicable but it is more expensive in both hardware cost and the software overhead in terms of communication and coordination. The range of MIMD architectures spans the range from the tightly coupled architectures such as multi-core CPU processors (shared memory MIMD architecture) and loosely coupled architectures such as clusters and warehouse-scale computers (distributed memory MIMD architecture) [4].

- ***MISD - Multiple instruction streams, single data stream***

  No commercial multiprocessors of this type have been built to date [4]. But they are applicable to specific scenarios such as pattern matching [7] or for redundant systems such as space flight controllers [8].

In the following, we discuss several important aspects in parallel architectures and variations of the traditional Flynn's taxonomy.

1.1.1 SIMD

The SIMD architecture is designed to exploit data-parallelism. The first SIMD instructions were essentially used to execute a vector of data with a single instruction through pipelined processors with a throughput of one word at a time. The first modern SIMD machines, e.g. Thinking machines CM-1 and CM-2, had thousands of limited-funtionality processors that would execute the same instruction on thousands of operand pairs simultaneously. Abundant data parallelism prevails in the context of scientific applications and graphics/video applications and their demands for higher performance are expected to grow especially in the era of big data.

SIMD support is now very common in modern CPUs and is used to improve performance for data parallel computations, real-time graphics processing, and digital signal processing. Such applications require the same type of operations to be repeatedly applied on a large amount of data. By amortizing the cost of decoding the common instruction and accessing, and processing the data elements with memory architecture friendly access pattern, a SIMD processor can achieve a speedup proportional to the vector size.

As early as 1996, MMX, a SIMD instruction set of integer operations, was introduced by Intel. A couple of years later, the 3DNow! [9] and SSE, SIMD instruction set extensions to the x86 architecture, were introduced by AMD and Intel for their processors (in 1998 and 1999 respectively). SSE targets single precision floating point operations using its designated register set (XMM registers) and was subsequently expanded by Intel to SSE2 through SSE4, achieving high popularity.

Advanced Vector Extensions (AVX), AVX2 and AVX-512 are the most recently

SIMD extension to the x86 instruction set architecture for microprocessors supported or announced from Intel and AMD. They support SIMD instructions of 128-bit/256-bit, 256-bit and 512-bit respectively along with a trend of the increase width. The AVX is supported on the Sandy Bridge and Ivy Bridge processors by Intel and on the Bulldozer, Piledriver processors and Trinity series APU from AMD. It features an increased register file width (128 bits to 256 bits), a three-operand SIMD instruction format and a new coding scheme that introduces a new set of code prefixes that extends the opcode space.

1.1.2 Multi-threading

Multi-threading is a technique that allows multiple streams of execution to take place concurrently within the same program, each stream processing a different transaction or message [10]. To begin with, we address two important concepts: thread and process. A process, typically created by the operating system, requires a significant amount of "overhead". Processes contain information about program resources and program execution state, including operating system related IDs, environment, working directory, registers, stack, inter-process communication tools, etc. Threads use and exist within these process resources, yet are able to be scheduled by the operating system and run as independent entities by duplicating only the bare essential resources that enable them to exist as executable code.

There are mainly three types of multi-threading: 1) block multi-threading, 2) interleaved multi-threading, and 3) simultaneous multi-threading. The first two types of multi-threading both belong to temporal multithreading. Block multi-threading refers to

the case that one thread runs until it is blocked by a long latency event when the OS switches in another ready-to-run thread. This type of multi-threading is also called cooperative or coarse-grained multithreading. Interleaved multi-threading [11] aims to reduce the data dependency stalls by enabling switching threads of execution on every clock cycle. Interleaved multi-threading result in multiple-context processors but such kind of processors require larger shared resources such as caches and TLBs to avoid thrashing between the different threads. On the other hand, for simultaneous multithreading, more than one thread can issue instructions on each cycle. Simultaneous multithreading is most notably applied to superscalar processors. Superscalar processors allow one thread to issue multiple instructions per cycle to explore instruction level parallelism. Simultaneous multithreading aims at increasing the utilization of processing resources.

Hyper-threading (HT) [12] is a classic example of simultaneous multithreading. Architecturally, an HT processor consists of two logical processors, each of which has its own processor architectural state, but sharing the execution resources. The architectural state resembles the context of a thread, including a number of data and control registers, and their own advanced programmable interrupt controller. On the other hand, the execution resources include the execution engine, the caches, the system-bus interface and the firmware. One key implication of HT is that from a software (OS's or programmers') perspective, the one physical processor appears to be two logical processors. Instructions from both threads are dispatched for execution by the execution source of the same physical processor core simultaneously. Out-of-order instruction scheduling is used to further explore instruction level parallelism. Due to the specific optimization target, namely, increasing the resource utilization, the actual performance scalability of applying hyper

threading or not is highly dependent on the nature of the application [10].

1.1.3 Graphics Processing Units (GPUs) and SIMT

The popularity of CPU-assisted real-time 3D graphics led to the development of hardware-accelerated 3D graphics, and eventually led to the popularity of the Graphics Processing Unit (GPU). Each of the multiprocessors of a GPU can be viewed as a SIMD architecture with substantially enhanced resources.

GPUs were introduced in 1999, replacing fixed-function graphics pipelines with fully programmable processors, ushering in the era of GPU-based high performance computation systems. By 2003, early pioneers were using graphics APIs to perform general purpose scientific calculations on GPUs. BionicFX, an audio processing company, used an NVIDIA GeForce 6800 card with their custom software architecture to "render" the data as needed. Since then, General Purpose GPUs, have been widely used to accelerate a wide range of applications.

On one hand, graphics processor providers have worked to develop GPUs specifically as general-purpose streaming processors, such as the NVIDIA Tesla series cards. On the other hand, GPGPU programming models and languages have evolved significantly. By 2007, NVIDIA addressed this need by introducing CUDA (Compute Unified Device Architecture), a general purpose parallel computing platform and programming model, which leverages the parallel compute engine in NVIDIA GPUs to allow efficient solutions for many complex computational problems.

To emphasize that CUDA is more flexible than the SIMD extension, for exam-

ple, allowing threads within the same warp to branch into different execution streams, NVIDIA extends the SIMD notion to SIMT (Single Instruction, Multiple Thread). The SIMT notion also emphasizes its hardware support of context switching of warps in the same thread block. Warp execution switching is a key CUDA technique to hide the high memory latency or any other high latency instructions. The high memory latency trades for the high memory bandwidth and less complicated memory caching mechanism.

A GPU consists of a number of streaming multi-processors and each streaming multi-processor contains a number of streaming processors. The execution runtime schedules thousands of threads in terms of thread blocks and dispatch blocks onto individual multi-processors based on resource requirements. Unlike CPUs being low latency, low throughput processors, GPUs are high latency, high throughput processors. Threads within a block are in turn organized as warps to be scheduled into execution. GPUs hide memory latency by switching stalling warps with ready-to-execute warps very fast. Thus, a good GPU application is able to hide the high memory latency while benefit from the high memory bandwidth and eventually give every high overall throughput.

Recent GPUs utilizing the CUDA programming model have attracted considerable interest in the high-performance computing community due to their extremely high peak performance, low cost, and the relative simplicity of the programming model. Moreover, these many-core processors tend to achieve much better performance-to-power ratios than the corresponding multicore CPUs. At the same time they enable scaling to thousands of cores on a single card. The power efficiency comes from the fact that processing instructions is actually expensive compared to floating point operations; using SIMD can amortize such overhead.

The CUDA programming model uses fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. Problems are partitioned into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem can be organized into finer pieces that can be solved cooperatively in parallel by all threads within the block [13]. The GPU executes data parallel functions called kernels using thousands of threads. A typical CUDA application achieves good performance by maximizing the utilization of the processor and memory resources.

1.1.4 MIMD

For MIMD computers, there are multiple processing elements, each of which has its own processor and memory. At each step, each processing element loads a separate instruction using its own program counter and a separate data element, applies the instruction to the data element, and stores a possible result back into memory. The processing elements work asynchronously relative to each other and communicate through an interconnection network. Clusters are typical examples of the MIMD model. Nearly all large scale parallel computers are based on the MIMD model, which can be further classified in terms of their memory organization and their interconnections. All supercomputers are clusters of MIMD processors with likely SIMD support in an individual processor and/or general purpose hardware accelerators. Memory can be organized as 1) distributed memory, 2) virtually shared memory or 3) shared memory. With distributed memory, each processor has its own physical memory and its own address space; with virtually shared

memory, the physical memory is typically distributed but there is a software layer that manages the overall memory as a single address space; with shared memory, all processors shared the same address space through hardware support.

From the programmer's point of view, memory can be viewed as distributed address space or shared address space. In the shared address space programming model, any processor can access any memory location, so its programming is comparatively easier relative to that of the distributed address space. The shared memory and the virtually shared memory make use of the shared address space and result the so-called UMA (Uniform Memory Access) and NUMA (Non-Uniform Memory Access). As the name indicates, UMA also requires that the access time from different processors to memory locations is the same while NUMA does not. However, the physically distributed property of the NUMA memory requires cache coherence between copies of a memory location, which makes the system harder to build.

Finally, for the distributed address space model, a processor can exchange information with another through message passing explicitly. One potential advantage of such systems is better potential scalability; however, achieving scalability of complex applications on a large system is usually non-trivial.

1.2 Parallel Programming Models

A parallel programming model presents an abstraction of the programming aspects of a parallel architecture or system. Such abstraction allows you to express concurrency and control flows of the objective application in particular ways. Parallel programming is

more complicated than sequential programming. There are several parallel programming models that are commonly used: 1) implicit model, 2) shared memory, 3) distributed memory, 4) data parallel and 5) task parallel, etc. [14]. We note that these programming models are not specific to a particular type of machine or memory architecture. In fact, any of these models can (theoretically) be implemented on any underlying hardware [14], though maybe unrealistically expensive.

1.2.1 Implicit Model

In an implicit model, the programmer does not need to take care of the concurrency or parallelism; instead, the compiler or interpreter would analyze the source code and extract the parallelism automatically based on the inherent language features. The implicit model is typically found in domain-specific languages where the concurrency is abundant for a typical application. Since the programmer is free of controlling parallelism manually, he/she can focus on expressing the algorithms. A pure implicitly parallel language does not require special directives, operators or functions to enable or guide parallel execution. HPF [15], LabVIEW [16], and MATLAB M-code [17] are typical examples of implicit parallelism. A frequent outcome of implicit parallelism is the less-than-optimal parallel efficiency, partially because the automatic parallelism is under-explored, also due to the fact that domain specific programmers tend to focus more on correctness rather than strive for performance.

1.2.2 Shared Memory Model

In a shared memory model, a program is basically a collection of threads having access to a shared memory. Communication is conducted implicitly by writing and reading shared variables, and special protection mechanisms such as locks, semaphores, atomic operations, and monitors are used to control concurrent access to shared resources. An advantage of this model is the relative straightforward programming concepts in terms of the data "ownership" - shared or private. The relatively low overhead of communication also allows very efficient utilization of the parallel computing resources. A handful of shared memory programming languages or systems are playing a significant role in high performance computing. Examples include Pthreads [18], OpenMP [19], Thread Building Blocks (TBB) [20], CILK [21], and Java threads [22]. A potential performance issue may result from the sophisticated memory hierarchy to ensure cache/data coherence. For example, it can incur significant overhead from cache refreshes and bus traffic when multiple processors are using the same data; this is especially severe for NUMA shared memory computers.

For shared memory models, the most prevalent theoretical model is the Parallel Random Access Machine (PRAM) model [23]. In the PRAM model, an arbitrary number of processors have access to an unboundedly large memory and operate synchronously on a shared input to produce some output. This is essentially a parallel version of the classic RAM model. The *Parallel Memory Hierarchy* (PMH) [24] model was proposed later, which uses a single mechanism to model the costs of both interprocessor communication and memory hierarchy traffic.

GPU programming models such as OpenCL and CUDA belong to the shared memory model. These shared memory programming models share similarities of underlying hardware in that they adhere to the PMH model of computation. Modern CPUs and GPUs all exhibit such a memory hierarchy where memory locality is typically exploited in an effort to match the processor performance. For example, memory hierarchies composed of registers, L1, L2 and L3 caches are typical in the current Intel and AMD's high-end processors such as the Intel Xeon E5 family processors. Registers, shared memory, L1, L2 caches and the global memory form the standard memory hierarchy of NVIDIA's most recent GPUs (e.g. Tesla K40).

A very important example of shared memory programming is OpenMP (Open Multiprocessing), which is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism in Fortran and C/C++ programs. Programmers specify a number of compiler directives to guide the parallelism. The compiler directives, library routines and environment variables together then determine runtime behavior. A master thread forks a number of parallelizing work threads according to the preprocessor directives, which then join back into the master thread after completion. Both task parallelism and data parallelism can be achieved using OpenMP in this way. The core elements of OpenMP are the constructs for thread creation, workload distribution (work sharing), data-environment management, thread synchronization, user-level runtime routines and environment variables [19].

Along with the simplicity and the productivity of the programming model of OpenMP, a notable drawback is that its lack of performance scalability in general. Another weakness is their limited expressiveness and that linear scalability is hard to achieve for a broad

application areas due to the high possibility of non-parallelizable tasks such as data dependency and non-parallelizable resources (most of which may be alleviated to a certain level given programmers' effort).

A more flexible example of the shared memory programming model is the POSIX Threads (Pthread), the POSIX (Portable Operating System Interface) standard API for creating and manipulating threads. Implementations of Pthread API are available on various platforms such as Unix-like operating systems such as GNU/linux, Mac OS X, etc, as well as Microsoft Windows. Most hardware vendors now support Pthreads in addition to their proprietary APIs. Pthreads are defined as a set of C language programming types and procedure calls, implemented with a pthread.h header file and a thread library. As all threads reside within the same address space as the process, which provide potentially efficient inter-thread communication. In many cases, programming with pthreads is easier to express the algorithms and can achieve optimum performance for shared memory processor architecture based application. Moreover, threaded applications enjoy practical advantages over non-threaded applications in several ways: 1) the CPU work can be overlapped with "time-consuming" I/O operations; 2) Priority or real-time tasks can be scheduled accordingly; and 3) asynchronous events can be handled by interleaved tasks. [18] However, a notorious pitfall for Pthread programming is the uneasiness of debugging in that data race bugs and deadlocks are easy to create and hard to locate.

1.2.3 Message Passing Model

In a message passing model, a parallel program consists of a number of cooperative processes, each with its own memory. Parallel processes exchange data through passing messages to one another. Process synchronization is done through waiting for messages to be delivered. Message passing systems can be either synchronous or asynchronous depending on the way the sender and receiver wait for the messages. Messages may have tags that can be used to sort messages. Two notable message passing models are the Message Passing Interface (MPI) and Parallel Virtual Machine (PVM) [25].

Message Passing Interface (MPI) is a message-passing library interface specification aiming at portability and ease of use. It is now a de facto standard specification for varied parallel architectures and is the dominant model used in the high-performance computing community. MPI is widely available including vendor-supplied implementations and royalty-free implementations. The standard defines the syntax and semantics of a core library routines useful to a wide range of users writing portable message-passing programs in Fortran and the C programming language. There has been three major generations of MPI standard: MPI 1.X, MPI 2.X and MPI 3.0. Popular MPI implementations include MPICH, MPICH2, Open MPI, LAM, MVAPICH2, etc. [26]

MPI is extremely portable - it is suitable for general MIMD or SPMD programs running distributed memory multiprocessors, networks of workstations, multi-core shared memory processors or a hybrid system [26]. In the beginning, MPI was designed for distributed memory architectures, which were becoming increasingly popular at that time. Later, shared memory SMPs were combined over networks, creating hybrid distributed

memory/shared memory systems. MPI implementations were extended to perform on both types of underlying memory architectures. However, the programming model is still a distributed memory model, despite the underlying physical architecture of the machine. In this programming model, the programmer is responsible for identify concurrency and implementing parallel algorithms using MPI constructs explicitly [27].

MPI standards includes the following: 1) Point-to-point communication, 2) Datatypes, 3) Collective operations, 4) Process groups, 5) Communication contexts, 6) Process topologies, 7) Environmental management and inquiry, 8) The info object, 9) Process creation and management, 10) One-sided communication, 11) External interfaces, 12) Parallel file I/O, 13) Language bindings for Fortran and C, and 14) Tool support [26].

The advantages of using MPI include the following [27]:

- ***Standardization –*** MPI is the de facto standard message passing library and is supported on virtually all HPC platforms. It has practically replaced all previous message passing libraries.

- ***Portability –*** There is no need to modify the source code when porting the application to a different platform that supports (and is compliant with) the MPI standard.

- ***Performance Opportunities –*** Vendor implementations should be able to exploit native hardware features to optimize performance.

- ***Functionality –*** A good number of routines are defined: over 115 routines in MPI-1 alone.

- ***Availability –*** A variety of implementations from vendors or from the public domain

are available.

PVM (Parallel Virtual Machines), on the other hand, is a set of software tools and libraries that emulate a general-purpose heterogeneous concurrent computing framework on interconnected computers of different architecture and operating systems. The objective is to enable utilization of such a collection of computers for concurrent or parallel computation. The individual computers can be shared- or local-memory multiprocessors, vector supercomputers, specialized graphics engines, or scalar workstations and PCs. The interconnection network is heterogeneous, such as Ethernet or FDDI. PVM consists of a run-time environment and library for message-passing, task and resource management, and fault notification. While PVM will not automatically make a commercial software package run faster, it does provide a cost effective way for large computational problems. PVM allows for world-wide distributed computing with tens of thousands of users [25].

1.2.4 Parallel Programming Abstraction and API

The learning curve and development lifecycle of parallel programs are two major obstacles for developers who desire to adopt a certain programming model. Therefore, in addition to these standard raw parallel programming models, and highly abstract parallel programming models, library APIs which free developers from keeping track of the underlying communication and synchronizations have emerged.

MapReduce [28] is probably one of the most popular parallel programming models in the era of big data. A MapReduce system normally consists of a large number of commodity machines and a specialized file system with a runtime framework to manage job

scheduling, synchronization, and communication. Users express solutions to real-world problems in terms of a number of map and reduce functions, and the runtime framework would take care of the actual execution. It aims at scalability for large scale problems.

The low cost, high performance, and energy-efficiency features of GPGPUs offer promising solutions for a wide range of real-world applications: Engineering/Manufacturing, Financial Services, Life Sciences, Entertainment and Digital Content Creation, Earth and Geo sciences, etc [29]. Various acceleration solutions such as OpenACC [30], PyCUDA [31], MATLAB GPU Computing [32], ArrayFire [33], etc allows easier access to NVIDIA's GPUs' extremely high processing capabilities in a relatively abstract way.

1.3 Supercomputing trends

1.3.1 History of Supercomputers

In the 1960s, Seymour Cray, Jim Thornton, and Dean Roush and about 30 other engineers built the CDC 6600 using silicon transistors made by Fairchild Semiconductor. With a relatively high speed clock and refrigeration cooling, the 6600 outran all computers of the time by about 10 times. It was dubbed a supercomputer and defined the supercomputer market when a number of CDC6600 were sold at $8 million each. This introduced the Cray-era of supercomputing, that spanned from mid-1970s to 1980s when a relatively small number (1-8) of vector processors were used to crunch numbers.

In the 1990s, the concept of massive parallelism was key to supercomputers, with thousands of processors connected by a high-speed network. In 1993, the Top500 [34] project was started. It ranks and details the 500 most powerful (non-distributed) computer

systems in the world twice every year at International Supercomputing Conference in June and Supercomputing Conference in November respectively. The yardstick is based on the Rmax from LINPACK MPP [35]. The Intel ASCI Red supercomputer, a mesh-based MIMD massively-parallel system with over 9000 computer nodes and well over 12 TB of disk storage, was the first system ever to break the 1TFLOPS barrier on the MP-Linpack benchmark in 1996. Significant progress has been made which lead to the introduction of more than 30 top supercomputers in the world which can perform more than 1 PetaFlops for the Top500 benchmark, according to the Nov 2013 list. The top 1 supercomputer NUDT Tianhe-2 from Guangzhou, China, is rated at 33.86 PetaFLops performance, almost twice as the top 2 Titan from Oak Ridge National Laboratory, in Tenneessee, USA.

1.3.2 Exascale Computing

Exascale computing refers to computing systems capable of at least one Exaflops (10^{18}), with a projected implementation by 2018 at SC'09.

1.3.2.1 Driving Applications

High-fidelity simulations of real-word systems constitute the greatest frontiers in computational physics, engineering and chemistry. Petascale computing opened the door for such real-world system simulations which currently suffer limitations in terms of temporal and spatial scales. "Predictive" science and engineering calculation requires Exascale computing capability to handle the complexity, for example, physical fidelity of

real-world systems [36].

Supercomputers have been widely used in various scientific and engineering areas, such as computational fluid dynamics, N-body simulations, weather forecast, reactor design simulation, bioinformatics and molecular dynamics, etc. Furthermore, with the Exascale computing, great transformation into high-fidelity simulation could take place in the following areas [36]:

- Aerospace, Airframes, and Jet Turbines
- Astrophysics
- Biological and Medical Systems
- Climate and Weather
- Combustion
- Materials Science
- Fusion Energy
- National Security
- Nuclear Engineering

1.3.2.2 Challenges of Exascale Computing

Exascale computing is expected to bring dramatic changes in high performance computing architectures as well as in software applications and algorithms. Furthermore, a key element of the strategy toward Exascale computing is the co-design of applications, architectures, and programming environments [1].

On the hardware side, the key issues are power and cooling constraints. The tradi-

tional Moore's law of doubling the clock speeds and transistor density every 18-24 months has been replaced by a doubling of cores/parallelism due to the power consumption, wire delays and cooling constraints, etc. On the other hand, the aggregated computing power of supercomputers requires hardware breakthrough to enable exascale computing later this decade, at least within any reasonable power budget [1]. The rule of thumb number for supercomputing power cost is around $1M per MW energy costs [1]. According to a 2013 DOE report [36], an exaflop system made entirely out of today's technology would probably cost $100B, requiring $1B per year to supply the needed power, and require its own dedicated power plant to produce that power. Therefore, to keep Total Cost of Ownership manageable, DOE's Exascale Initiative Steering Committee adopted 20MW as the upper limit for a reasonable system design (movable but at great cost and design risk) with a platform capital cost under $200M.

The Green500 List provides some technology paths towards exascale computing: heterogeneous supercomputing systems totally dominates the top 10 spots on the Nov 2013 release list. A heterogeneous system uses computational building blocks that consists of traditional multi-core CPUs, general purpose GPUs and/or co-processors. Heterogenous systems of Exascale will have chips with thousands of tiny processor cores and a few large ones. This is due to power consideration and core functionality and better on-chip memory bandwidth (avoids chip pin limit). However, even projecting the top 1 Green supercomputer TSUBAME-KFC's energy efficiency to exascale, the extrapolation to an exaflop supercomputer would be 222MW, well beyond from the DOE's target of 20-MW system power envelope [37].

Projecting from the current technology, by 2018 it is expected to be easy to put

10 Teraflops on a single chip that consumes 100W. But to supply the processing units with modest memory bandwidth to floating point ratio of 0.2, it would require 2000W power for the memory. To meet such power budget target, it is expected that hardware breakthroughs including memory subsystems, 3D memory, 3D packaging, large-scale optics based interconnects, etc. are expected to take place [1].

On the software side, architecture-awareness algorithm and software design is a key issue. Achievable performance per watt will likely be the primary measure of progress. Data movement is expensive, in the sense of power consumption and application performance. Table 1.1 shows an approximate power costs of different data operations credited to John Shalf [1] and minimizing data movement and performing more work per unit data movement is critical in future algorithms and software design.

Table 1.1: Approximate Power Costs (in picoJoules) [1]

	2011
DF FMADD flop	100 pJ
DP DRAM read	4800 pJ
Local Interconnect	7500 pJ
Cross System	9000 pJ

Specific critical issues and features at Petascale and Exascale algorithm and software design include the following [38]:

- ***Reduce the synchronization and communication***

  The traditional fork-join model generates choke points at the join and wastes cycles. It makes sense to break the traditional fork-join model or build data-dependence based synchronization. Communication avoidance algorithms [39] are more per-

formance friendly in such a massively parallel environment. At a higher level, it is more appealing to use methods which have lower bound on communication.

- ***Using mixed precision methods***

  Most processors that are targeting scientific computations execute twice flops as fast for single precision as double precision operations. The 2x data size difference between them also indicates 2x speed difference for data movement.

- ***Auto tuning or adaptive algorithms***

  Automatic performance tuning uses machine time instead of human time for tuning to find the optimal or optimum execution plan by searching over possible implementations. Atlas (BLAS) [40], FFTW [41], Sprial(DSP) [42], PhiPAC(BLAS) [43], etc are some of the most popular auto tuned libraries. In the era of Exascale computing, architectures would be much more complicated and we would resort to auto tuned programs using smart search space trimming. At the very least, auto-tuning could ease code generation for new architectures and possibly help software developers to learn the new architectures. Aside from the auto-tuned libraries and subroutines, additional adaptive runtime could be used to resolve the precedence-constraints as necessary to avoid wasted cycles during synchronization [44].

- ***Fault resilient algorithms***

  The chance of component failure grows with the size of the system. Today, Sequoia BG/Q node failure rate is 1.25 failures/day; with 1000x processing elements increase, runtime errors would be much more frequent, and necessary measures are required to identify and correct such errors.

- ***Hierarchical Programming Model***

  It is unlikely to support globally flat bandwidth across a system for Exascale without major breakthroughs in packaging technology or photonics. Moreover, due to the heterogeneity of the future system, a hybrid of multi-core, many-core, and massively parallel accelerating cores, different types of parallelism needs to be distinguished in a programming environment. Hierarchical parallel programming model (rather than the flat MPI or shared memory/PRAM model) that allows algorithm designers to express and control data locality, data flow and parallel granularity and hierarchy is necessary [1].

We have discussed above the Exascale computing challenges from the hardware and software perspective separately. The application-driven design process aims to finding the best technology to run the code; on the other hand, the technology driven design process is to fit your application to the technology. Either solution is sub-optimal to the best possible achievable performance. Co-design of applications, architecture, and programming environment was proposed as one key element to meet the Exascale challenges. It is believed to be an unprecedented opportunity for application and algorithm developers to influence the direction of future architectures so that the meet DOE mission needs [1].

A living example of the good performance of co-design is the development of Anton [45] - a massively parallel supercomputer designed and built by D. E. Shaw Research. It is a special purpose system for molecular dynamics simulations of proteins and other biological macromolecules. The building blocks of the system are two subsystem based specialized ASICs to deal with two different calculations with different underly-

ing physics mechanisms for the simulation. The resulting system runs several orders of magnitude faster and made previously impossible simulations possible.

1.3.3 GPU or CPU-GPU heterogeneous clusters

The massively-parallel hardware architecture and high performance of floating point arithmetic and memory operations on GPUs match the requirements of computational demanding scientific computing applications, leading to the wide use of GPU accelerators on HPC clusters. Such clusters are superior in terms of space, power, cooling demands and reduced number of operating system images that must be managed relative to traditional CPU-only clusters of similar aggregate computational power [46].

The NVIDIA's Tesla series GPUs and Intel's Xeon Phi Coprocessors are two of the most frequently used accelerators for the top supercomputers on the Top500 list. For example, the top 1 supercomputer Tianhe-2 uses Intel Xeon Phi 31S1P Coprocessors and the top 2 supercomputer Titan uses NVIDIA Tesla K20x cards. Another notable fact is that all the top 10 supercomputers on the Green500 list are accelerated by NVIDIA Tesla K20x/K20m GPU.

The scale of the clusters also varies from large-scale supercomputers with more than 18K nodes as in Titan [47], to relatively smaller scale clusters with 16 nodes as in [48]. A larger amount of work have been reported using various GPU or CPU-GPU based clusters in recent years where CUDA applications are extended into multiple GPUs using OpenMP, MPI, and other standard parallel programming framework. Most of the cluster based work try to optimize the inter-node interconnection network/communication and/or

CPU-GPU memory transfer to improve the performance when communication is the bottleneck [49], [48]. Other work seeks algorithmic improvement by altering algorithms using the fixed-digit representation of floating-point data as long as the necessary/precision is preserved [50] [51].

With the increasing variety of GPU acceleration solutions, OpenCL [52] has been designed for general purpose programming for GPUs as a platform-independent programming model. It is available on most platforms including NVIDIA/AMD/ARM GPUs, Intel/AMD's multi-core CPUs, as well as Intel's MIC architecture. However, OpenCL requires performance-reducing initializations that do not exist in other languages such as CUDA. In [53], Du et. al pointed out in 2011 that while the Khronos group developed OpenCL with programming portability in mind, performance is not necessarily portable.

To simplify parallel programming of heterogeneous CPU/GPU systems, companies including Cray, NVIDIA, PGI and CAPS developed a new parallel programming standard - OpenACC (Open Accelerators) in 2011. It bares similarity of OpenMP which allows programmers to provide simple "directives" to the compiler for parallelism. The OpenACC API describes a collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator. OpenACC is expected to be complementary to existing HPC programing models such as OpenMP, MPI, CUDA and OpenCL. The target users are scientists interested in accelerators who can benefit from a simpler programming model and organizations with a significant investment in legacy production applications that have not yet been parallelized. [54]

1.4 Mathematical Background

1.4.1 Fast Fourier Transform

The Fourier Transform and its discrete version, the Discrete Fourier Transform (DFT), constitute some of the most fundamental tools used throughout science and engineering. The introduction of the Cooley-Tukey [55] Fast Fourier Transform (FFT) algorithm is considered to be a breakthrough that has led to a number of very efficient methods for computing the DFT. These methods have enabled the widespread use of the FFT algorithm by both practitioners and researchers in a wide range of science and engineering applications such as computational fluid dynamics and digital signal processing. Since its introduction, considerable efforts have been devoted to map the FFT computation onto various specialized and general purpose parallel architectures, as they emerged over the years, so as to enable computational scientists to handle larger and larger scale applications.

1.4.1.1 FFT Algorithms

The one-dimensional discrete Fourier transform of n complex numbers represented by an array X is the complex vector represented by the array Y defined by:

$$Y[k] = \sum_{j=0}^{n-1} X[j]\omega_n^{jk} \tag{1.1}$$

where $0 \leq k < n$, and $\omega_n = e^{\frac{-2\pi\sqrt{-1}}{n}}$ the n^{th} root of unity. Various Fast Fourier Transform (FFT) algorithms have been proposed since the early 1960's, each of which has computational complexity of $O(n \log n)$. The most famous FFT algorithm is the Cooley-Tukey algorithm that uses a divide-and-conquer strategy to decompose a large size DFT into smaller size DFT's and compute these DFT's recursively. More specifically, let $n = n_1 n_2$ and let $j = j_1 n_2 + j_2$ and $k = k_1 + k_2 n_1$ for $0 \leq j, k < n$ with $0 \leq j_1, k_1 < n_1$, and $0 \leq j_2, k_2 < n_2$. Then Eq (1.1) can be re-written as:

$$Y[k_1 + k_2 n_1] = \sum_{j_2=0}^{n_2-1} \left[\left(\sum_{j_1=0}^{n_1-1} X[j_1 n_2 + j_2] \omega_{n_1}^{j_1 k_1} \right) \omega_n^{j_2 k_1} \right] \omega_{n_2}^{j_2 k_2} \tag{1.2}$$

Eq (1.2) expresses the DFT computation as a sequence of three steps. The first step consists of n_2 DFT's each of size n_1, called radix-n_1 DFT, and the second step consists of a set of twiddle factor multiplications (multiplications by $\omega_n^{j_2 k_1}$). Finally, the third step consists of n_1 DFTs each of size n_2, called radix-n_2 DFT.

The Cooley-Tukey algorithm can be implemented in a number of ways depending on the recursive structure and the input/output order. Two important variations based on the recursive structure are the so-called the *decimation in time (DIT)* and the *decimation in frequency (DIF)* algorithms. The *DIT* algorithm uses n_2 as the initial radix, and recursively decomposes the DFTs of size n_1; while the *DIF* algorithm uses n_1 as the initial radix, and recursively decomposes the DFTs of size n_2. In this thesis, we will focus on the *DIF* algorithm.

We note the two variations regarding the input and output orderings, namely in-order and bit-reversed order. Assuming that all the steps are carried out in-place, an exam-

ination of Eq(1.2) indicates that, after the first step, the output array becomes $XA[k_1 n_2 + j_2]$, and after the twiddle factor multiplication step, the output array is $XB[k_1 n_2 + j_2]$, while after the 3^{rd} step, the output array becomes of the form $XC[k_1 n_2 + k_2]$. A quick comparison against the DFT output array $Y[k_1 + k_2 n_1]$ implies that if both the radix-n_1 and radix-n_2 DFTs are in order, we would need a transposition of the intermediate output array after the 2^{nd} step so that the output is in order. However, if both the radix-n_1 and radix-n_2 DFTs are computed in bit-reversed order (namely, direct butterfly execution), and no transposition is done after the 2^{nd} step, we would generate a size n DFT with bit-reversed order output. In this thesis, we will use the in-order input, bit-reversed order output since the corresponding in-place computation will allow us to better exploit the characteristics of the global memory. However, our algorithm can be converted to the in-order input, in-order output version accordingly.

A multi-dimensional DFT can be defined recursively as a set of DFTs along each of the dimensions of a multi-dimensional array. In particular, the 3D DFT of a 3D array of size $I \times J \times K$ is defined as follows:

$$Y[i,j,k] = \sum_{n=0}^{K-1} \sum_{m=0}^{J-1} \sum_{l=0}^{I-1} X[l,m,n] \omega_I^{il} \omega_J^{jm} \omega_K^{kn} \tag{1.3}$$

For each element in the DFT array, it is a summation of all the input elements multiplied by a specific coefficient determined by the input and output indices. Clearly, the order of the dimensions can be arbitrary, and the computational can be carried out in any order of the dimensions. Applying the Cooley-Tukey algorithm along each dimension, we can compute the 3D FFT on N elements in $O(N \log N)$ complexity.

1.4.2 Tridiagonal Solver

A tridiagonal solver handles a system of n linear equations of the form $Ax = d$, where A is a tridiagonal matrix, x and d are vectors. This can be represented in matrix form by:

$$\begin{bmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ 0 & & & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_n \end{bmatrix}$$

A simplified form of Gaussian elimination, called Thomas' algorithm, is a well-known classical algorithm to solve this problem. The algorithm consists of two sweeps: forward elimination and backward substitution. The forward sweep updates both the vectors b and d, and the backward substitution determines the unknown vector x.

```
for (int i = 1; i < n; i++)
{
        double m = a[i]/b[i-1];
        b[i] = b[i] - m*c[i-1];
        d[i] = d[i] - m*d[i-1];
}
```

Listing 1.1: Forward Elimination

```
x[n-1] = d[n-1]/b[n-1];

for (int i = n - 2; i >= 0; i--)

        x[i]=(d[i]-c[i]*x[i+1])/b[i];
```

Listing 1.2: Backward Substitution

We make the following observations regarding Thomas' algorithm.

- The complexity of the algorithm is $O(n)$, and the algorithm as described seems to be inherently sequential.
- Four one-dimensional arrays for the input a, b, c and d are needed in the general case.
- It may appear that we need an array for the output x vector; however, the unknown vector can be stored in the d vector during the backward substitution step.

1.4.3 Poisson Equation and Background

Projection methods are very effective in solving time-dependent incompressible flow problems [56]. The general algorithm is based on a Helmholtz decomposition of the velocity vector field and typically consists of two stages: in the first stage, an intermediate velocity that does not satisfy the incompressibility constraint is computed at each time step; in the second stage, the pressure is used to project the intermediate velocity onto a space of divergence-free velocity field. To facilitate the latter, a Poisson equation for the pressure is solved. In most cases this is a computationally expensive step, which takes a large fraction of the CPU time per time step, and therefore is critical to the performance of the overall solver.

Projection methods have been extensively used in high-fidelity computations of turbulent and transitional flows, where eddy resolving techniques, such as Direct Numerical Simulations (DNS) and large-eddy simulations (LES) are utilized. In most of the early DNS/LES, building-block problems such as turbulent boundary layers and shear layers

had been considered in simple geometrical configurations, where structured Cartesian grids and compact discretization stencils can be used (see for example [57] and [58] for reviews). Critical to the success of these simulations was the development of parallel and efficient solvers for the Poisson equation. In most cases fast direct solvers were utilized, based on FFT transforms [59], cyclic reduction [60] and their combination [61], Divide & Conquer [62] and several other variances.

The need to simulate more complex flows, where boundary-fitted grids (i.e unstructured) are utilized, shifted the attention to iterative methods, which are able to deal with the complexity of the resulting algebraic systems. Significant effort has been made to optimize these methods on leadership high-performance computing platforms (see for example [63]). On the other hand, the effort to further advance direct solvers was significantly less due to the relatively narrow area of applications, at least in fluid mechanics related problems. Recently, however, with the advent of immersed-boundary (IB) methods (see [64] for a recent review) there is a renewed interest for highly efficient direct solvers for structured Cartesian grids. In IB methods the requirement for the grid to conform to the body is relaxed and boundary conditions are imposed using a specially designed forcing function. As a result complex moving boundaries can be treated using highly efficient structured solvers eliminated the need for grid regeneration/adaptation.

In the following we illustrate the mathematical formulation of the FFT-based Direct Poisson Solver and the induced solvers on two different types of boundary conditions (BC).

1.4.3.1 FFT-based Direct Poisson Solver

Let us consider the three-dimensional Poisson equation,

$$\nabla^2\phi = \frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} + \frac{\partial^2\phi}{\partial z^2} = f, \text{ in } \Omega, \tag{1.4}$$

discretized using second-order, central finite difference approximations on a Cartesian grid:

$$\begin{aligned} \nabla^2\phi &\approx \frac{1}{\Delta x^2}(\tilde{\phi}_{i+1,j,k} - 2\tilde{\phi}_{i,j,k} + \tilde{\phi}_{i-1,j,k}) + \frac{1}{\Delta y^2}(\tilde{\phi}_{i,j+1,k} - 2\tilde{\phi}_{i,j,k} + \tilde{\phi}_{i,j-1,k}) \\ &+ \frac{1}{\Delta z^2}(\tilde{\phi}_{i,j,k+1} - 2\tilde{\phi}_{i,j,k} + \tilde{\phi}_{i,j,k-1}) = \tilde{f}_{i,j,k} \end{aligned} \tag{1.5}$$

By means of Fourier transformation the three-dimensional problem gets reduced to a system of one- dimensional Helmholtz equations, which have to be solved for each Fourier mode. We will consider two different boundary condition configurations in Ω, as discussed in the following sections.

1.4.3.2 Three-Periodic Boundary Conditions

For all points on the computational grid we replace $\tilde{\phi}_{i,j,k}$ and $\tilde{f}_{i,j,k}$ by their 3D DFT decompositions to obtain:

$$\frac{1}{IJK}\sum_{n=0}^{K-1}\sum_{m=0}^{J-1}\sum_{l=0}^{I-1}\Big(\frac{2}{\Delta x^2}\left(cos(2\pi l/I)-1\right)\hat{\phi}_{l,m,n}+\frac{2}{\Delta y^2}\left(cos(2\pi m/J)-1\right)\hat{\phi}_{l,m,n}$$
$$+\frac{2}{\Delta z^2}\left(cos(2\pi n/K)-1\right)\hat{\phi}_{l,m,n}\Big)e^{2\pi i\frac{il}{I}}e^{2\pi i\frac{jm}{J}}e^{2\pi i\frac{kn}{K}}$$
$$=\frac{1}{IJK}\sum_{n=0}^{K-1}\sum_{m=0}^{J-1}\sum_{l=0}^{I-1}\hat{f}_{l,m,n}e^{2\pi i\frac{il}{I}}e^{2\pi i\frac{jm}{J}}e^{2\pi i\frac{kn}{K}}, \tag{1.6}$$

Hence, in the frequency domain, the above system of equations reduces to a diagonal linear system of the form:

$$\left(\frac{2}{\Delta x^2}\left(cos(2\pi l/I)-1\right)+\frac{2}{\Delta y^2}\left(cos(2\pi m/J)-1\right)+\frac{2}{\Delta z^2}\left(cos(2\pi n/K)-1\right)\right)\hat{\phi}_{l,m,n}=\hat{f}_{l,m,n} \tag{1.7}$$

which can be solved very efficiently. In the present study this step will be fully integrated into the FFT step to avoid global memory overhead. The diagonal entries are defined as follows:

$$D_{l,m,n}=D_l+D_m+D_n$$

where,

$$D_l=2I^2\Big[cos(2\pi\frac{l}{I})-1\Big],\quad D_m=2J^2\Big[cos(2\pi\frac{m}{J})-1\Big],\quad D_n=2K^2\Big[cos(2\pi\frac{n}{K})-1\Big] \tag{1.8}$$

We refer to $D_{l,m,n}$ as scalars and to D_l, D_m and D_n as subscalars. Note that the scalars are uniquely determined by the l, m, n indices and the grid size. The procedure to handle the three periodic boundary conditions case can be described as follows:

- Compute the 3D Fast Fourier Transform of the 3 dimensional source dataset $\tilde{f}_{i,j,k}$ to generate $\hat{f}_{l,m,n}$.
- Divide each $\hat{f}_{l,m,n}$ by the correspondent scalar $D_{l,m,n}$ to get the 3 dimensional unknown dataset $\hat{\phi}_{l,m,n}$.
- Compute the 3D Fast Inverse Fourier Transform of the new 3 dimensional unknown dataset $\tilde{\phi}_{i,j,k}$ to obtain the solution.

1.4.3.3 The Two-Periodic, One-Neumann Boundary Conditions Case

Let us assume that periodic boundary conditions are imposed the X and Y directions, while Neumann boundary conditions apply in Z. As in the above case, if we replace $\tilde{\phi}_{i,j,k}$ and $\tilde{f}_{i,j,k}$ by their DFT decompositions along X and Y we get:

$$\begin{aligned}
&\frac{1}{IJ}\sum_{m=0}^{J-1}\sum_{l=0}^{I-1}\Big(\frac{2}{\Delta x^2}\left(cos(2\pi l/I)-1\right)\hat{\phi}_{l,m,k}+\frac{2}{\Delta y^2}\left(cos(2\pi m/J)-1\right)\hat{\phi}_{l,m,k}\\
&+\frac{1}{\Delta z^2}\left(\hat{\phi}_{l,m,k+1}-2\hat{\phi}_{l,m,k}+\hat{\phi}_{l,m,k-1}\right)\Big)e^{2\pi i\frac{il}{I}}e^{2\pi i\frac{jm}{J}}\\
&=\frac{1}{IJ}\sum_{m=0}^{J-1}\sum_{l=0}^{I-1}\hat{f}_{l,m,k}e^{2\pi i\frac{il}{I}}e^{2\pi i\frac{jm}{J}}
\end{aligned}\tag{1.9}$$

For each coefficient, we have

$$\frac{1}{\Delta z^2}\hat{\phi}_{l,m,k+1} + 2\left(\frac{1}{\Delta x^2}(cos(2\pi l/I) - 1) + \frac{1}{\Delta y^2}(cos(2\pi m/J) - 1) - \frac{1}{\Delta z^2}\right)\hat{\phi}_{l,m,k} + \frac{1}{\Delta z^2}\hat{\phi}_{l,m,k-1} = \hat{f}_{l,m,k} \tag{1.10}$$

The above expressions yield $I \times J$ tridiagonal linear systems, each of which involves K equations. Note that the coefficients in the same tridiagonal linear system are determined by their l, m indices and the grid size. Equations with the same $[l, m]$ pairs are dependent and belong to the same linear system while those with different $[l, m]$ pairs are independent and belong to different linear systems. The overall algorithm can be described as follows:

- For each value of k, $0 \leq k \leq (K-1)$, compute the 2D forward Fast Fourier Transform on the corresponding slice of the 3 dimensional source dataset $\tilde{f}_{i,j,k}$ to get $\hat{f}_{l,m,k}$.
- Solve the $I \times J$ tridiagonal linear systems (with size $K \times K$ coefficient matrices) to get $\hat{\phi}_{l,m,k}$.
- For each value of k, compute the 2D inverse Fast Fourier Transform on the corresponding slice of the 3 dimensional unknown dataset $\tilde{\phi}_{i,j,k}$.

1.5 Major Contributions of This Thesis

In this dissertation, we focus on developing optimization techniques of mapping algorithms and applications on to CUDA GPUs and CPU-GPU heterogeneous platforms

for a number of demanding scientific applications. More specifically, we start by tacking multi-dimensional FFTs computation on single GPUs, then we integrate the FFT kernels into a FFT-based Poisson solver on a single GPU. We then study the more practical cases, CPU-GPU heterogeneous platform. We develop a software pipeline based scheme to port the Poisson solver onto the heterogeneous platform. In the last, we extend the software pipeline based scheme onto another core computation - DGEMM.

First, we address the problem of mapping three-dimensional FFTs onto a number of CUDA GPUs. We exploit the high-degree of multi- threading offered by the CUDA environment while carefully managing the multiple levels of the memory hierarchy in such a way that: (i) all global memory accesses are coalesced into 128-byte device memory transactions issued in such a way as to optimize effects related to partition camping, locality, and associativity. and (ii) all computations are carried out on the registers with effective data movement involved in shared memory transposition. In particular, the number of global memory accesses to the entire 3-D dataset is minimized and the FFT computations along the X dimension are almost completely overlapped with global memory data transfers needed to compute the FFTs along the Y or Z dimensions.We were able to achieve performance between 135 GFLOPS and 172 GFLOPS on the Tesla architecture (Tesla C1060 and GTX280) and between 192 GFLOPS and 290 GFLOPS on the Fermi architecture (Tesla C2050 and GTX480). The bandwidths achieved by our algorithms reach over 90 GB/s for the GTX280 and around 140 GB/s for the GTX480.

Second, we develop a highly multithreaded FFT-based direct Poisson solver on CUDA GPUs. We carefully decompose the direct Poisson solver into a number of procedures and align and order the computation of (X, Y, and Z) to best suited to the memory hi-

erarchy of the GPUs. In addition, we integrate some procedures together to minimize the times of global memory access. Also we mix the single-precision and double-precision by only boost the precision when necessary such that the global storage of single-precision is necessary but guarantee the second-order accuracy of the solver. As a result, we have achieved up to 140 GFLOPS and a bandwidth of 70 GB/s on the Tesla C1060, and up to 375GFLOPS with a bandwidth of 120GB/s on the GTX 480. The performance of our algorithms is superior to what can be achieved using the CUDA FFT library in combination with well-known parallel algorithms for solving tridiagonal linear systems of equations.

Next, we extend our high performance FFT-based direct Poisson solver on CPU-GPU heterogeneous platforms for the case when the input is too large to fit on the GPU global memory. Our scheme is consisted of a number of dependent techniques that work together for an overall superior performance. First of all, the overall solver is decomposed to be CPU-part work and GPU-part work, which the CPU-part work would not only suited to the CPU's strength but also tacked certain dependence for the resultant GPU work to be a number of independent tasks. Then, based on our software pipeline, those independent tasks are transferred from the CPU main memory to the GPU device memory through the PCIe bus. This software pipeline is optimized in such a way that the PCIe bus transfer time of one task is overlapped with some other task's execute time in the GPU. The overall effect is that only one PCIe bus memory transfer is used and the effective bandwidth is optimal while the suitable part of the application is accelerated by the GPU. We were able to achieve significantly better performance than what has been reported in previous related work, including over 145 GFLOPS for the three periodic boundary conditions (single precision version), and over 105 GFLOPS for the two periodic, one Neumann boundary

conditions (single precision version). The effective bidirectional PCIe bus bandwidth achieved is 9-10 GB/s, which is close to the best possible on our platform. For all the cases tested, the single 3D data PCIe transfer time, which constitutes a lower bound on what is possible on our platform, takes almost 70% of the total execution time of the Poisson solver

Finally, we extend our software pipeline on the CPU-GPU heterogeneous platforms to a computational-bounded algorithm - double precision matrix multiplication (GEMM). Similarly, we address the case when the input is too large to fit onto the GPU global memory. We adapt the blocking algorithms into a strategy that achieves near peak GPU computational rate within the bandwidth constraint of the PCIe bus bandwidth. By ensuring contiguous and near-peak- rate kernel execution flows, we were able to achieve more than 1 and 2 TFLOPS performance on a single node with dual socket multicore CPU using 1 and 2 GPUs respectively. Our results suggest the possibility of developing matrix computations on heterogeneous platforms which achieve native GPU performance on very large data sizes up to the capacity of the CPU memory.

Chapter 2: Multi-dimensional FFT on GPUs

In this chapter, we address the problem of mapping three-dimensional Fast Fourier Transforms (FFTs) onto the recent, highly multithreaded CUDA Graphics Processing Units (GPUs) and present some of the fastest known algorithms for a wide range of 3-D FFTs on the NVIDIA Tesla and Fermi architectures. We exploit the high-degree of multithreading offered by the CUDA environment while carefully managing the multiple levels of the memory hierarchy in such a way that: (i) **all** global memory accesses are coalesced into 128-byte device memory transactions issued in such a way as to optimize effects related to partition camping [65], locality [66], and associativity. and (ii) all computations are carried out on the registers with effective data movement involved in shared memory transposition. In particular, the number of global memory accesses to the entire 3-D dataset is minimized and the FFT computations along the X dimension are almost completely overlapped with global memory data transfers needed to compute the FFTs along the Y or Z dimensions. We were able to achieve performance between 135 GFlops and 172 GFlops on the Tesla architecture (Tesla C1060 and GTX280) and between 192 GFlops and 290 GFlops on the Fermi architecture (Tesla C2050 and GTX480). The bandwidths achieved by our algorithms reach over 90 GB/s for the GTX280 and around 140 GB/s for the GTX480.

Table 2.1: Basic Parameters of the Four Evaluated GPUs

	SMs	SPs/SM	Regs	Shared Mem	Global Mem	Mem BW	Clock Freq
Tesla C1060	30	8	16K	16KB	4GB	102GB/s	1296MHz
GTX280	30	8	16K	16KB	1GB	141.7GB/s	1296MHz
Tesla C2050	14	32	32K	48KB[1]	3GB	144GB/s	1147MHz
GTX480	15	32	32K	48KB[1]	1.5GB	177.4GB/s	1401MHz

2.1 CUDA GPU Overview

Recent GPUs using the CUDA programming model have attracted considerable interest in the high-performance computing community due to their extremely high peak performance, low cost, and the relative simplicity of the programming model. Moreover these many-core processors tend to achieve much better performance to power ratios than the corresponding multicore CPUs while at the same time scaling to thousands of cores on a single card. The CUDA programming model uses multi-threading and data parallelism to exploit the many-core architectures of the recent NVIDIA GPUs, thereby achieving orders of magnitude better performance compared to multicore CPUs, especially on scientific applications. In this section, we start by giving an overview of such architectures, focusing on the four platforms used in our tests, followed by a summary of the main features of the CUDA programming model. We pay a particular attention to the memory model since this will play a central role in our algorithms.

The basic architecture of the recent NVIDIA GPUs consists of a set of Streaming Multiprocessors (SMs), each of which containing up to 32 Streaming Processors (SPs or cores) executing in a SIMD fashion; a large number of registers; and a small shared memory organized into multiple banks. Threads running on the same SM can share data

[1]The shared memory size of the two Fermi devices is the default size

and synchronize, limited by the available resources (number of registers and size of the shared memory) on the SM. Each GPU has small constant and texture caches (typically around 64KB). All the SMs have access to a very high bandwidth Global Memory; such a bandwidth is achieved only when simultaneous accesses are coalesced into contiguous 16-word lines. However the latency to access the global memory is around 400-800 cycles, which is quite high. A summary of the parameters of the four platforms we use in this paper is given in Table 2.1.

The CUDA programming model envisions phases of computations running on a host CPU and a massively data parallel GPU acting as a co-processor. The GPU executes data parallel functions called kernels using thousands of threads. Each GPU phase is defined by a grid consisting of all the threads that execute some kernel function. Each grid consists of a number of thread blocks such that all the threads in a thread block are assigned to the same SM. Several thread blocks can be executed on the same SM, but this will limit the number of threads per thread block since they all have to compete for the resources (registers and shared memory) available on the SM. Programmers need to optimize the use of shared memory and registers among the thread blocks executing on the same SM, if any.

Each SM schedules the execution of its threads into warps, each of which consists of 32 parallel threads. For the Tesla architecture (16 banks), a shared memory request for a warp is issued in two memory requests, one for each half-warp with a speed of two clock cycles. On the other hand, for the Fermi architecture (32 banks), a shared memory request for a warp is issued in one memory request with a speed of two clock cycles. When all the operands of the warps are available in the shared memory, the SM issues a

single instruction for the 16 threads in a half-warp. The cores within an SM will be fully utilized as long as operands in the shared memory reside in different banks of the shared memory (or access the same location from a bank). If a warp stalls, the SM switches to another warp resident in the same SM.

Optimizing performance of multithreaded computations on CUDA requires careful consideration of global memory accesses (as few as possible and should be coalesced into multiple of contiguous 16-word lines); shared memory accesses (threads in a warp should access different banks); and partitioning of thread blocks among SMs; in addition to carefully designing highly data parallel implementations for all the kernels involved in the computation. In particular, threads in a half-warp which access contiguous words in the global memory are grouped together into a single coalesced global memory access thereby achieving the best possible throughput. Otherwise CUDA uses the minimum number of coalesced global memory accesses to cover the region touched by the half warp.

2.2 Our Overall Strategy and Core Techniques

Our work is based on the *DIF* version of the original Cooley-Tukey algorithm with in-order input and bit-reversed order output. A key feature of this algorithm is the "in-place" computation for all stages of the computation, which we will exploit to use memory access patterns that achieve good memory bandwidth. Our scheme targets large size 3D FFT such that no dimension is smaller than 128 as long as the input data can fit in the device memory. Every data element is assumed to be a complex number such that each of

the real and imaginary parts is a single-precision floating point number, and hence each complex number is represented by 8 consecutive bytes. Our implementations are tuned to both the Tesla and Fermi architectures, which turn out to require slightly different implementations but with the same overall approach.

2.2.1 Representation of the 3D FFT Decomposition

As noted before, the Cooley-Tukey algorithm to compute the DFT of $n = n_1 \times n_2$ elements consists of three steps, the first of which involves n_2 radix-n_1 DFTs, followed by twiddle factor multiplications, and ending with n_1 radix-n_2 DFTs. Since we are dealing with 3D data, we need to specify the decomposition for computing the DFT along each dimension, as well as the data sets used for each radix computation. We will represent such a decomposition by making use of the tensor representation originally introduced in FFTW.

We first note that the data elements of a 3-D array will be stored in the device memory along the X dimension first, then the Y dimension followed by the Z dimension. Consider for example an array of size $256 \times 256 \times 256$. The entries of each vector along the X dimension will appear as a contiguous block of 256 complex numbers, while the entries of a vector along the Y dimension will have a stride of 256 between any consecutive entries of the vector. Along the Z dimension, consecutive entries will be 256×256 entries apart on the device memory. The FFT computation along each dimension will be specified by a number of FFTs each with a possibly different radix and each operating on the data along the dimension using a stride relative to *that dimension*. The actual global

memory stride can easily be computed from such a specification. More specifically, a decomposition say $n = n_1 \times n_2$ (that is, radix-n_1 followed by radix-n_2) along the X dimension will be represented as follows:

- $X(n_2, n_1, n_2, n, tw)$
- $X(n_1, n_2, 1, n_2, no-tw)$

The above representation should be interpreted as follows. We start by performing n_2 FFTs each of radix n_1 on data along the X dimension with stride n_2, and hence these FFTs encompass n entries, followed by twiddle factor multiplications (which in our case are computed on the fly using fast intrinsic sine/cosine functions provided by CUDA). Then n_1 FFTs, each of radix n_2 is computed on the data along the X dimension with a stride of 1, and hence each FFT encompasses n_2 contiguous elements. We can extend the same representation to a decomposition with more factors such as $n = n_1 \times n_2 \times n_3$. Assuming that n is the size of the X dimension, this decomposition can be represented as:

- $X(n_2 n_3, n_1, n_2 n_3, n, tw)$
- $X(n_1 n_3, n_2, n_3, n_2 n_3, tw)$
- $X(n_1 n_2, n_3, 1, n_3, no-tw)$

The use of dimension name (X in the above equation) is necessary since we will be interleaving the radix computations between the different dimensions. Let's consider for a simple example the case of $256 \times 256 \times 256$ where the decomposition along the X dimension is given by $256 = 16 \times 4 \times 4$, while the decompositions along the Y and Z dimensions are identical $256 = 16 \times 16$. One (extremely inefficient) way to compute the corresponding 3D FFT can be represented as follows:

- $X(16, 16, 16, 256, tw)X(64, 4, 4, 16, tw)X(64, 4, 1, 4, no-tw)$
- $Y(16, 16, 16, 256, tw)Y(16, 16, 1, 16, no-tw)$
- $Z(16, 16, 16, 256, tw)Y(16, 16, 1, 16, no-tw)$

2.3 CUDA Architecture Constraints

In this section we outline our main strategies to map the FFT computation on the Tesla and Fermi architectures so as to optimize the use of the available resources (both computation and memory resources) while managing the constraints imposed by these architectures.

2.3.1 Managing the CUDA Memory Hierarchy

The CUDA memory hierarchy consists of a global memory accessible by all the streaming processors, coupled with a shared memory and a set of registers on each of the SMs. Given that the FFT computation involves operations along each of the dimensions over a large 3D dataset stored in global memory, we have to pay a particular attention to the memory hierarchy while trying to execute a highly multithreaded computation.

Given the typical size of our FFT computations, all the input, intermediate, and output data have to be held in the global memory, which has the largest access latency (400-800 cycles) in the memory hierarchy. Global memory accesses are carried out as 32-byte, 64-byte, or 128-byte device memory transactions. To achieve high bandwidth, global memory accesses must be coalesced - that is, global memory loads and stores by a half thread warp must be contiguous so as to result in a very few (one if possible) mem-

ory transactions. Since each complex number in our computation is represented by 8 bytes, aligned consecutive memory access of threads of a half-warp satisfies the largest 128-byte memory transaction size. In fact, global memory accesses issued by the threads in a warp will be executed as two 128-byte device memory transactions on either architecture thereby achieving a very good memory bandwidth. Unlike previously published GPU FFT algorithms, we always ensure coalesced 128-byte global/device memory transactions in addition to exploiting spatial and temporal locality to optimize effective device memory bandwidth. In particular, we exploit low-level device memory system hardware features to approach the theoretical device memory bandwidth. Device memory partition [65] and memory locality [66] are two important issues for a very good bandwidth. For example, the device memory of GTX280 has 8 partitions and hence active warps should avoid issuing transactions that touch only a subset of them (so-called partition camping). Row access locality of device memory [66] is also preferred for high memory bandwidth, which can be interrupted by both algorithm restrictions and memory access streams issued by active warps. Note that the performance bottleneck of a relatively optimized radix FFT kernel is still the effective global/device memory throughput and hence we focus on memory optimization.

Compared to the global memory, the shared memory is much faster. The size of the shared memory per SM is 16KB for compute capability 1.3 (GTX280 and Tesla C1060) and 48KB (the default size) for compute capability 2.0 (GTX480 and Tesla C2050). Note that the shared memory size of the Fermi architecture can be configured between 16 KB and 48 KB. Each shared memory is divided into equal-sized memory modules (banks) so as to enable concurrent access. For the Tesla architecture, the bank count is 16 (half-warp)

and for the Fermi architecture, the bank count is 32 (warp). The shared memory access is most efficient when bank conflicts are avoided, and hence we developed a general bank conflict free data transposition strategy. We observe that the L1 cache available on the Fermi architecture does not seem to significantly speed-up our FFT implementations while the L2 cache plays an important role.

Registers represent the fastest level of the memory hierarchy and are allocated to live threads; the peak arithmetic throughput can only be achieved by using registers rather than the shared memory [67]. The total number of 32-bit registers available is 16KB for compute capability 1.3 and 32 K for compute capability 2.0. We note that a thread is allocated at most 128 registers for compute capability 1.3 and 64 registers for compute capability 2.0 even though the compute capability 2.0 SM has more registers overall. The number of registers available and the maximum number of registers that can be allocated to a thread will have a direct impact on the radix decomposition adopted for each size. In particular, the maximum number of registers that can be allocated to a single thread on the Tesla architecture allows us to compute a radix-32 FFT using only the registers, which cannot be done on the Fermi architecture. For the latter architecture, we have to use more than a single thread to compute a radix-32 FFT. *In our implementation, an FFT of any radix along X, Y or Z dimension is computed directly on the registers, with the FFT computations along the X dimension almost completely overlapped with global memory data transfers needed to compute the FFTs along the Y or the Z dimension. This constitutes a major feature of our algorithms which distinguishes it from other published algorithms.*

2.3.2 Managing CUDA Threads

Note that CUDA programs rely on thread parallelism to hide memory and arithmetic latencies. However, relying only on increasing thread parallelism to optimize performance is not necessarily a good strategy because of the limits on several hardware resources such as number of registers and size of shared memory. Based on our experience, 64 threads per block on the Tesla architecture and 128 threads per block on the Fermi architecture seem to achieve the best balanced performance. In addition, we try to overlap global data movement and small radix computations along the X dimension to alleviate the latency dependency with the relatively small thread block parallelism. Our strategy is to make each thread compute a relatively small size FFT directly and use more threads to compute a single radix FFT if necessary. We will explain this process further later.

2.4 Overall Strategy

In our implementation, each kernel loads and stores the entire 3D data once from and into the global memory during which FFTs of certain radix sizes are carried out along possibly two dimensions concurrently. In general, we attempt to overlap a small radix FFT computation along the X dimension with data movement from the global memory needed for FFT computations along other dimensions. The mathematical properties of the Cooley-Tukey algorithm provide a rich set of possibilities for decomposing and reordering the overall computation so as to exploit the main characteristics of either the Tesla or Fermi architecture.

We start by stating an immediate implication of the mathematical formulation of the Cooley-Tukey FFT algorithm related to the ordering of the FFT subcomputations.

- Given a decomposition of the FFT along each dimension into a series of small-radix FFTs, each of which to be called an *FFT sub-computation*, we can arbitrarily inter-mix the FFT sub-computations of different dimensions as long as the relative ordering of the FFT sub-computations along each dimension is preserved.

This property was also observed by Gu et al. [68] .

We are now in a position to provide the main features of our strategy.

- The FFTs along the Y and the Z dimension are computed through separate kernels (typically two kernels for each dimension) while the FFT sub-computations along the X dimension are inserted into the kernels corresponding to the Y and Z dimensions. Occasionally, the FFT sub-computations along the Y and the Z dimension may be combined in the same kernel for improved performance on the Tesla architecture.

- The kernels to execute the FFT sub-computations along the Y and Z dimensions achieve high-bandwidth global memory accesses through the coalesced access of chunks of contiguous 128-bytes (16 elements) along the X dimension and through tuning the memory transactions issue sequence for device memory locality optimization. The corresponding radix FFTs are computed directly on registers.

- The FFT sub-computations along the X dimension are computed during the execution of the kernels for the Y and Z dimension FFT computations through the use

of the shared memory to transpose data across the registers while avoiding bank conflicts.

- Within each kernel, the data loading (from global memory or shared memory re-arrangement) and the FFT sub-computations are organized in such a way that the dependency between the data supply and the computations is optimized to match the execution pipeline.

The implementation of this strategy consists of three main steps. The first amounts to decomposing appropriately each of the Y and Z dimension size into a product of radixes (typically two) each of which is handled by a kernel. The second step involves a decomposition of the X dimension, taking into consideration the decompositions along the Y and Z dimensions. At this step, we need to figure how to insert each of the corresponding FFT sub-computations along X into one of the Y or Z kernels so as to achieve high memory bandwidth and overlapped computation and data movement. Finally, we have to determine the workload of each thread and allocate the appropriate number of threads to each FFT radix computation. We will next describe the strategy to carry out each of these steps using the case of $256 \times 256 \times 256$ on the Tesla architecture.

2.4.1 Y and Z Dimension Decomposition

Two main factors seem to play a dominant role in determining the best decomposition for each of the Y and Z dimensions. Given that each Y or Z FFT sub-computation will access memory in a coalesced manner along the X dimension, the available resources have to be able to support a batch of 16×2^k Y and Z FFT sub-computations in the X

dimension in parallel, for some non-negative integer k. The second factor is to try to achieve a load balance between different kernels while ensuring overall effective global memory access by the kernels.

The first factor puts an upper bound on the size of the radix that can be used on a given architecture, and the second implies almost balanced decomposition for each of the Y and Z dimensions whenever such a decomposition is needed.

Consider our running example of an input of size $256 \times 256 \times 256$. Since we won't be able to accommodate 16 FFT(256) on a single SM of Tesla (which is usually the case for large size Y/Z dimension transform), each of the Y and Z dimensions has to be decomposed into a product of radixes. A balanced decomposition suggests that we use $256 = 16 \times 16$ for each of the Y and Z dimensions, implying the following four FFT sub-computations along the Y and Z dimensions:

- $\{Y(16, 16, 16, 256, tw)\}$
- $\{Y(16, 16, 1, 16, no{-}tw)\}$
- $\{Z(16, 16, 16, 256, tw)\}$
- $\{Z(16, 16, 1, 16, no{-}tw)\}$

Braces are used to indicate the boundaries of each kernel. We will next describe how to insert the FFT sub-computations along X into these kernels in such a way that their executions will be almost completely overlapped with the coalesced memory accesses for the above kernels.

2.4.2 X Dimension Decomposition

As we move data from the global memory in a coalesced fashion to carry out the FFT sub-computations along Y and Z, we organize each of the X dimension transforms into smaller-radix FFTs that can be incorporated into the kernels executing the Y and Z FFT sub-computations. Therefore the data movement should be organized so that each of the FFT sub-computations along X can be carried out by the same thread block executing the kernels. However, our Cooley-Tukey algorithm (*DIF* version) requires larger strides in early stages and smaller strides in later stages while the coalesced global memory access requires consecutive accesses to contiguous 128×2^k bytes of data. We resolve this tension between these requirements by using a number of small contiguous chunks with some stride in the X dimension for the earlier stages while using a large contiguous chunk for the later stages. Loading the data through the use of multiple small chunks (each chunk is of size 128 bytes) will incur a certain performance degradation, which depends on the size of the strides. In general, the FFT along X dimension is decomposed into three or four small-radix FFTs such as radix-2, radix-4, or radix-8 FFTs.

Consider again our running example of $256 \times 256 \times 256$ data size whose FFT has to be computed on a Tesla GPU. We decompose the X dimension as $256 = 4 \times 8 \times 8$ and hence each such FFT can be computed as the sequence:

- $X(64, 4, 64, 256, tw)$
- $X(32, 8, 8, 64, tw)$
- $X(32, 8, 1, 8, no-tw)$

Suppose we want to insert the first FFT sub-computation into a Y kernel, which

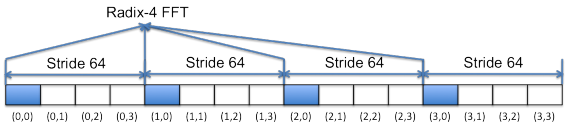


Figure 2.1: X Dimension Element Partition

implies 64 sets of radix-4, stride 64 computation with associated twiddle factors for each row of 256 elements. For the Tesla architecture, we use a 64-thread block to load 64 elements for the X dimension in one row and exchange elements using block synchronization. To accommodate the computation and performance requirement, 256 elements in a row are partitioned into 4×4 sub-groups each of size 16 denoted from (0, 0), (0, 1) up to (3, 3) accordingly. This partition imposes a stride-64 (Figure 2.1) between elements of the same sub-group index from (0, x), (1, x), (2, x) and (3, x). Then 4 blocks of 64 threads consisting of 16 half-warps will be responsible for the 16 sub-groups and 4 half-warps from the same thread block will access the corresponding sub-group (0, x), (1, x), (2, x) and (3, x), (x can be 0, 1, 2, 3 for 4 blocks). Note sub-group data chunks are each of size 128-byte, namely the maximum coalesced device memory transaction size. Finally our overall algorithm for computing FFT($256\times256\times256$) can be summarized by the following representation in which each kernel is enclosed between braces.)

- $\{Y(16, 16, 16, 256, tw)\}$
- $\{X(64, 4, 64, 256, tw), Y(16, 16, 1, 16, no-tw)\}$
- $\{X(32, 8, 8, 64, tw), Z(16, 16, 16, 156, tw)\}$
- $\{X(32, 8, 1, 8, no-tw), Z(16, 16, 1, 16, no-tw)\}$

We will later provide the details about how the various small-radix FFTs are allocated to the thread blocks. Since each of the last three kernels contains FFT sub-computations along two distinct dimensions, the intermediate data needs to be appropriately transposed

through the shared memory so that the corresponding FFT sub-computations can be carried out effectively. This is explained next.

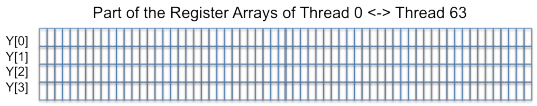
2.4.3 Bank Conflict Free Shared Memory Transposition

The FFT sub-computations along the Y and Z dimensions are always carried out directly on registers. To compute a small-radix FFT along the X dimension, we have to use the shared memory to transpose the data and move it back into registers before completing the sub-computations, after which we have to transpose back the elements into the registers as in the original layout for further processing.

To make efficient use of the shared memory, bank conflicts have to be avoided, although occasionally, trading bank conflicts for smaller shared memory usage can actually result in better performance. This will occur in some kernels on the Fermi architecture.

Additional requirements on the shared memory transposition include balanced workload and avoiding warp divergence among the threads in a thread block.

The word size of each bank is 32-bit, the same size of a register and half the size of a complex number. To avoid bank conflicts, we separate the transposition of the real parts and the imaginary parts and add padding as necessary. We only consider the real parts for now; the imaginary parts are handled in a similar way. The transpose operation is carried out more or less the same way on both the Tesla and the Fermi architectures. At the beginning, the elements held in the registers are transferred into the shared memory and then loaded back in a transposed fashion into the registers. After the X dimension radix computation, a reverse transpose is conducted through the shared memory to restore



(a) Register Arrays of 64 threads;

Shared Memory Array of Size 16x16

[0, 0~15]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
[0, 16~31]	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
[0, 32~47]	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
[0, 48~63]	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
[1, 0~15]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
[1, 16~31]	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
[1, 32~47]	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
[1, 48~63]	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
[2, 0~15]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
[2, 16~31]	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
[2, 32~47]	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
[2, 48~63]	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
[3, 0~15]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
[3, 16~31]	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
[3, 32~47]	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
[3, 48~63]	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

(b) Store Elements from Registers to the Shared Memory Array

Shared Memory Array of Size 16x16

[0, 0~15]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
[0, 16~31]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
[0, 32~47]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
[0, 48~63]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
[1, 0~15]	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
[1, 16~31]	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
[1, 32~47]	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
[1, 48~63]	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
[2, 0~15]	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
[2, 16~31]	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
[2, 32~47]	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
[2, 48~63]	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
[3, 0~15]	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
[3, 16~31]	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
[3, 32~47]	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
[3, 48~63]	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

(c) Load Elements from the Shared Memory to Registers

Figure 2.2: Shared Memory Transposition

the original layout of the data.

Continuing with our $256 \times 256 \times 256$ example and focusing on the second kernel above, we use 64 threads to load a 64×16 sub-array along the $X \times Y$ dimensions such that each half-warp loads four 128-byte chunks along the X dimension each time, for a total of 16 times load, ending up with each thread holds 16 Y dimensional elements with stride 16 in the end. Note that to ensure full utilization of the threads and maintain balanced workloads, each thread will have to compute 4 sets of radix-4 FFT along the X dimension and one set of radix-16 FFT along the Y dimension in four execution loops; namely, each time 4 rows of 64 elements are transposed. The data layout in the registers

is illustrated in (Figure 2.2a) where the column corresponding to thread i represents the data held in the registers allocated to that thread. Our goal is to"transpose" this initial data layout so that it is stored into the shared memory as illustrated in (Figure 2.2b) and is loaded from the shared memory as illustrated in (Figure 2.2c).

Bank conflicts occur when multiple threads try to access different words from the same bank. The Tesla architecture has 16 banks and in this transposition scenario, bank conflicts do not occur. In other cases, we may have to use padding. Consider for example the case when we have to perform $X(8, 8, 8, 64, tw)$, namely, the workload of one block from the 4 blocks computing one row of $X(32, 8, 8, 64, tw)$. In this case, we need an 8×64 shared memory to transpose so that each thread will have its 8 elements required by the radix-8 FFT along the X dimension. This time, upon loading, every 8 consecutive threads will load 8 times of 8 consecutive elements from each 64-element row. Since 64 is a multiple of the number of banks (16), the number of banks used in the first row will need to be shifted in the second row to avoid threads in two consecutive rows trying to access the same bank. Namely, we need to pad 8 elements per 64-element row in this step and hence the resulting shared memory is of size $8 \times (64 + 8)$.

In general, the key idea is to stagger the banks from row to row so that bank conflicts are avoided. By using this strategy, it is clear that we will always be able to avoid bank conflicts.

2.4.4 Execution Plans

Once we have decided on the sequence of kernels to be executed, we have to allocate the operations to threads, which have to be organized into thread blocks and grid blocks.

It turns out that we use more or less fixed-size thread blocks for each of the Tesla and Fermi architectures. More specifically, we typically use 64 threads per block on the Tesla and 128 threads per block on the Fermi. We assign operations to thread blocks in such a way as to optimize the device memory throughput with respect to the partition camping problem and the row locality issue. We use a 2D representation $\{x_{size}, y_{size}\}$ for each block. The x_{size} is used to represent the number of threads along the X dimension, for each fixed value of X. The y_{size} is used to represent the number of threads used to compute the radix-FFT sub-computations along either the Y or the Z dimension. Therefore the total number of threads in a block is $x_{size} \times y_{size}$. Clearly the x_{size} threads are allocated to handle the X-dimension FFT sub-computations as well as transposition.

The organization of the grid of thread blocks is managed as a 2D array [x, y]. The x dimension of the array corresponds to the number of blocks used to cover the X dimension of the input data. For example, if the X dimension FFT size is 256 and the number of the threads in a block is 64, then we should have 4 blocks for the X dimension. The y dimension of the grid corresponds to the number of blocks in Y and Z dimension. For our running example, the first kernel of the Y dimension needs 16 of $256/16$ blocks to cover the data plane corresponding to a single Z coordinate value. To cover the entire data set, we need 16×256 blocks. We may change to a more balanced execution declaration (i.e. 4×16 as the x vector and 256 as the y vector) to avoid the CUDA grid size declaration

limit. The thread blocks are executed in the order of their block IDs, so the block ID assignment should be tuned to optimize the device memory throughput, mainly for locality. For the Y dimension sub-steps, assigning block ID according to memory layout {x, y, z} is in general quite good.

We end this section by stating a couple of optimization techniques that may need to be applied to achieve optimized device memory throughput.

- *In-place or out-of-place execution*. Our algorithm is an in-place algorithm (reading and writing with the same stride), which helps to manipulate the memory access pattern. However occasionally, we may want to exploit out-of-place execution order (options) for global memory accesses locality possibility. Out-of-place execution for Y and Z dimension involves transposition in Y/Z dimension between sub-steps of the same dimension transform (which is merely a different stride access of device memory among rows (X dimension) of 256 elements). Such transposition can be done together during the storing into and the loading from the global memory step and results in global memory access with a balanced stride among kernels for the same dimension FFT computation. Whether it is actually adopted needs to be tuned with specific data sizes. Take size 256 FFT in the Y dimension for example. It is decomposed into 16x16. An in-order execution will consist of (i)16 sets of radix-16 with input and output stride 16 with twiddle, (ii) 16 sets of radix-16 with input and output stride 1. However, an out-of-order execution will consist of two 16 sets of radix-16 with input stride 16 and output stride 1, in addition to the twiddle multiplications. Note that we only tune this execution order for Y and Z dimension

for the overall global memory latency while the properties of the memory access in the X dimension are all preserved.

- *Intermediate memory for smaller memory stride in Z dimension transform*. Based on the algorithm, the strides of the Z dimension are much larger than the Y dimension; if Z dimension transform is computed in more than one kernel, strictly implemented from the algorithm will yield relatively large global memory latency. This optimization attempts to make use of device memory transaction locality. We believe such an approach will provide more opportunities to achieve better device memory bandwidth throughput.

- *Ordering of Y and Z dimensions*. We always compute the Y dimension transform before the Z dimension. Inserting the X dimension sub-steps will involve stride-coalesced global memory access. Inserting such an X dimension access stride into the Y dimension kernels is much smaller than that the corresponding Z dimension kernels. Also, sub-steps of the same dimension matter when the sizes are not the same. For example, if we decompose Y dimension FFT size 128 into 16x8, which radix to compute first matters because this results in different memory strides. This probably arises from different pipeline granularities of continuous device memory transaction issues and computation workload of the same thread.

2.5 Performance Evaluation

The performance of our 3D FFT scheme is evaluated on four NVIDIA GPU cards: two Tesla architecture cards with compute capability 1.3 (GTX280 and Tesla C1060), and two Fermi architecture cards with compute capability 2.0 (GTX480 and Tesla C2050). Hence for each architecture we have two cards with similar execution units but different memory bandwidths. Specifically, the GTX280 and the Tesla C1060 have the same number of identical streaming multiprocessors with respectively 141GB/s and 102GB/s peak device memory bandwidths. For the other two variations of the Fermi architecture, the peak device memory bandwidths are respectively 144GB/s (Tesla C2050) and 177GB/s (GTX480).

In our tests, the size of each dimension of the 3D FFT is a power of two and all of our implementations have been carefully compared to the output produced by CUFFT for correctness.

We capture two performance measures: the number of GFlops and the global memory bandwidth by our implementations. More precisely, if the execution time of our 3D FFT on data of size $NX \times NY \times NZ$ is t seconds, then its GFlops is measured using the standard formula:

$$GFlops = \frac{5 \cdot NX \cdot NY \cdot NZ \cdot [\log_2 (NX \cdot NY \cdot NZ)] \cdot 10^{-9}}{t} \tag{2.1}$$

Regarding the effective global memory bandwidth achieved, we use the formula:

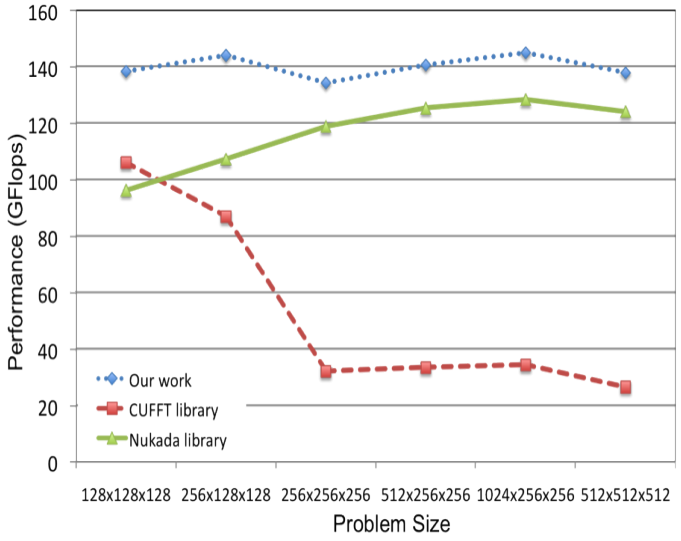
$$BW = \frac{8 \cdot NX \cdot NY \cdot NZ \cdot \#\_of\_accesses \cdot 10^{-9}}{t} \tag{2.2}$$

where the #_of_accesses is the total number of global memory accesses (loading or storing). Each of our tests (our algorithm and other libraries as available) is run 5 times after which the arithmetic mean of the total runtime is used to compute the performance measures introduced above.

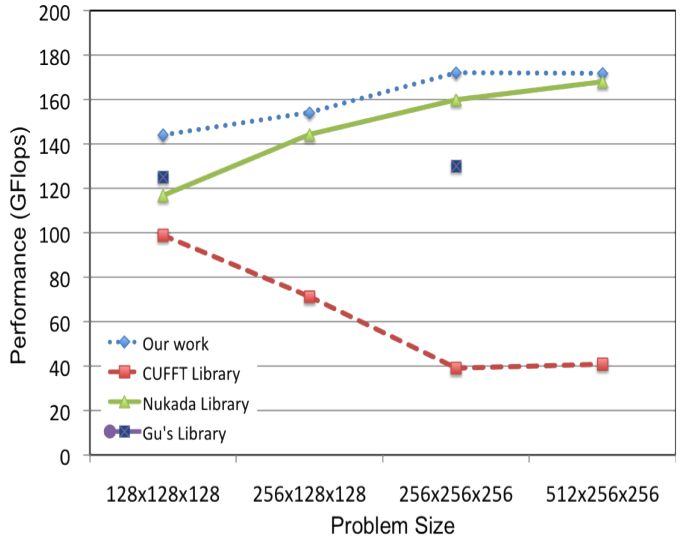
2.5.1 Performance Evaluation on the Tesla Architecture

Figure 2.3a illustrates the performance of our algorithm on the Tesla C1060 card compared to the best previous algorithms, and Figure 2.3b illustrates the corresponding performance on the GTX280. For each case, we try to increase the 3D data size up to the maximum possible that can fit into the global memory of the device. We run the tests using our algorithm, the CUFFT library, and the Nukada Library [69]. For Gu's performance on GTX280, we extracted the numbers from their paper [68]. The detailed decomposition, grouping and ordering schemes used for our implementations are given in the appendix. It is clear that our strategy achieves significantly better performance than the previous known schemes. Detailed execution plans can be found in [70].

In our implementations, we used the same programs for the Tesla architecture, except for the data size $256 \times 128 \times 128$. We slightly re-tuned the $256 \times 128 \times 128$ directly on the GTX280. As mentioned earlier, we expect better performance on the GTX280 since the theoretical bandwidth increases from 102GB/s to 141.7GB/s. The performance for



(a) Performance Comparison on Tesla C1060



(b) Performance Comparison on GTX280

Figure 2.3: Performance Evaluation on Tesla-based GPUs

the original code on data of size $256\times128\times128$ is respectively 144 GFlops and 140 GFlops on the Tesla C1060 and the GTX280. In the initial code, we decompose each of the Y and Z dimension transforms into 32×4 and 4×32 and combine the radix-4 sub-steps from the two dimensions into one kernel, inserting the X dimension transforms into kernels. This results into a relatively significant computation workload for each kernel, including large radix FFTs and transpositions. Such workload allocation is favored by the Tesla

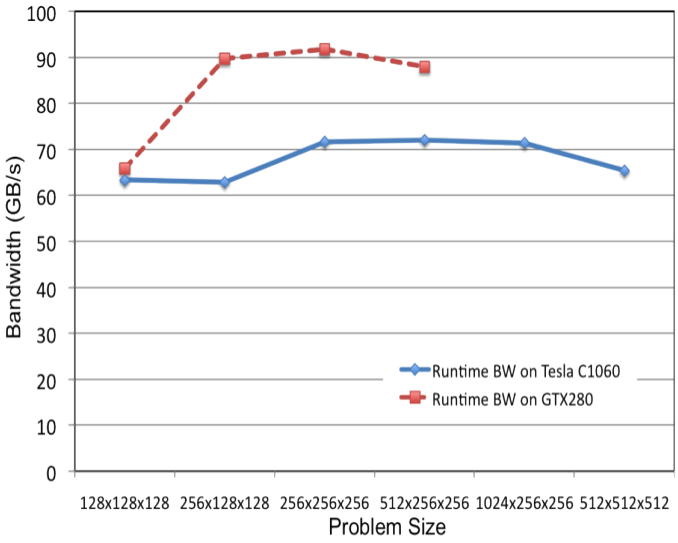


Figure 2.4: Actual Bandwidth on Tesla Devices

Table 2.2: Bandwidth achieved on the Tesla architecture cards

Data size	BW on Tesla C1060	BW on GTX280
128x128x128	63.27 GB/s	65.85 GB/s
256x128x128	62.86 GB/s	89.70 GB/s
256x256x256	71.64 GB/s	91.76 GB/s
512x256x256	72.02 GB/s	87.92 GB/s
1024x256x256	71.39 GB/s	NA[1]
512x512x512	65.37 GB/s	NA[1]

C1060 since the overhead of the device memory latency is much more significant (around 30%) than that of the GTX280. The code for $128 \times 128 \times 128$ is the same because of its competitive performance on both cards; the computation overhead is not as significant as that of $256 \times 128 \times 128$ since the X dimension size is smaller.

Figure 2.4 and Table 2.2 show the actual bandwidth utilization of our implementations. As we can see from the figure, the actual device memory bandwidth of Tesla C1060 is usually lower than that of the GTX280 except for the computation-bound data size ($128 \times 128 \times 128$).

[1] "NA" indicates cases of memory size usage larger than the global memory capacity.

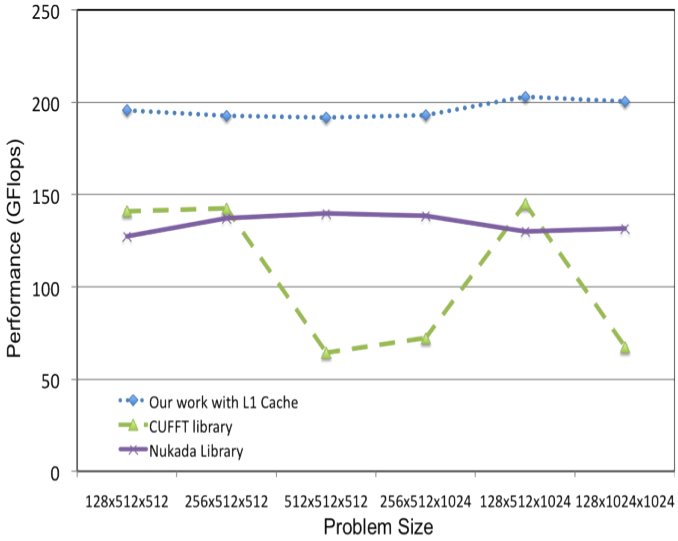


Figure 2.5: Performance Evaluation on Tesla C2050

2.5.2 Performance Evaluation on the Fermi Architectures

Figures 2.5 and 2.6 illustrate the performance of our algorithms on the Tesla C2050 and the GTX480, compared to the best known 3D FFT algorithms on these platforms. Figure 2.7 illustrates the actual global memory bandwidth achieved on the two Fermi devices. The numbers reported were obtained by running our algorithms, the CUFFT library, and the Nukada library [69], on the same size 3D datasets. Detailed execution plans can be found in [70].

Similarly, we use the same code, initially tuned on Tesla C2050, and evaluate the performance on both cards. Hence we are able to achieve around 200 GFlops on the C2050 and above 260 GFlops on the GTX480. We note the possibility of using caching on Fermi by setting the compilation flag on L1 and L2 cache. According to [71] all accesses to GPU DRAM go through L2, including CPU-GPU memory copies. For Fermi devices, global memory accesses are cached: the compilation flag -dlcm is used to determine if it

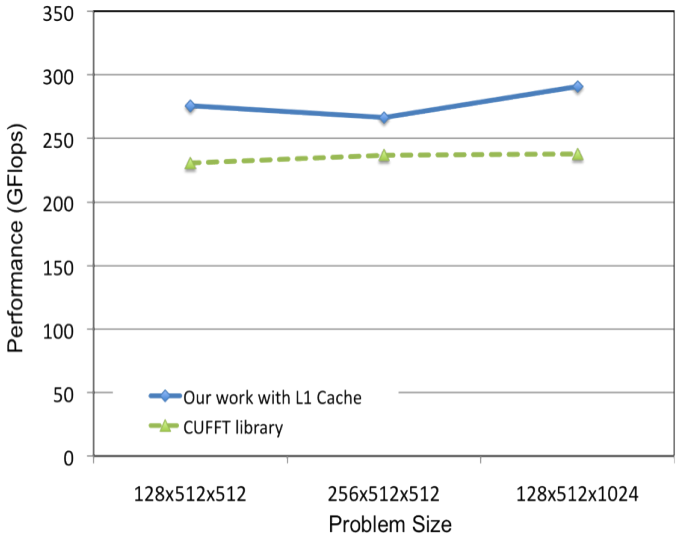


Figure 2.6: Performance Evaluation on GTX480

can be cached in both L1 and L2 (the default setting) (dlcm=ca) or in L2 only (dlcm=cg). We evaluate the performance difference of caching effects on the two cards and is shown in Table 2.3 and Table 2.4. The evaluation indicates the L1 cache does not help much.

Table 2.3: Cache Effects of Performance on Tesla C2050

Data size	Tesla C2050 with L1+L2 Cache	Tesla C2050 with L2 Cache only
128x512x512	195.70 GFlops	195.42 GFlops
256x512x512	192.69 GFlops	200.97 GFlops
128x512x1024	202.97 GFlops	203.04 GFlops
512x512x512	191.70 GFlops	195.04 GFlops
256x512x1024	193.07 GFlops	191.04 GFlops
128x1024x1024	200.37 GFlops	201.60 GFlops

Table 2.4: Cache Effects of Performance on GTX480

Data size	GTX480 with L1+L2 Cache	GTX480 with L2 Cache only
128x512x512	275.58 GFlops	284.88 GFlops
256x512x512	266.42 GFlops	280.05 GFlops
128x512x1024	290.83 GFlops	289.53 GFlops

Figure 2.7: Actual Bandwidth on Fermi Devices

Table 2.5: Actual Bandwidth on Fermi Devices (GB/s)

Data size	C2050 L1+L2	C2050 L2	GTX480 L1+L2	GTX480 L2
128x512x512	100.20	100.06	141.10	145.86
256x512x512	94.86	98.94	131.14	137.87
128x512x1024	99.92	99.96	143.18	142.53
512x512x512	91.01	92.47	NA	NA
256x512x1024	91.53	90.57	NA	NA
128x1024x1024	94.99	95.58	NA	NA

Table 2.5 shows the actual bandwidth utilization of our implementations with both L1 and L2 cache and just with L2 cache. As expected, the actual device memory bandwidth achieved on the Tesla C2050 is lower than that of the GTX480.

Chapter 3: FFT Based Poisson Solver on a Single GPU

In this chapter, we present a highly multithreaded FFT-based direct Poisson solver that makes effective use of the capabilities of the current Graphics Processing Units (GPUs). Our algorithms carefully manage the multiple layers of the memory hierarchy of the GPUs such that all the global memory accesses are coalesced into 128-byte device memory transactions, and all computations are carried out directly on the registers. A new strategy to interleave the FFT computation along each dimension with other computations is used to minimize the total number of accesses to the 3D grid. We illustrate the performance of our algorithms on the NVIDIA Tesla and Fermi architectures for a wide range of grid sizes, up to the largest size that can fit on the device memory ($512 \times 512 \times 512$ on the Tesla C1060 and Tesla C2050 and $512 \times 256 \times 256$ on the GTX 280 and GTX 480). The performance of our algorithms is superior to what can be achieved using the CUDA FFT library in combination with well-known parallel algorithms for solving tridiagonal linear systems of equations.

3.1 Optimized GPU Implementation for the Case of Three Periodic BC

3.1.1 Overall Strategy

The basic approach was described earlier and consists of a 3D forward FFT, scaling of each element, and a 3D inverse FFT. The scaling of each element during the intermediate step depends only on the three-dimensional indices of the element. Therefore we can use any combination of DIF or DIT, in-order or bit-reversed order, FFT algorithm as long as we can easily track the indices of each element. In our implementation, we modify the small radix-k FFT no-twiddle codelets generated by genfft from FFTW [41] with twiddle factor formula for convenience.

Recall the register pressure on the radix FFT size and the shared memory size influence over the maximum FFT size in the X, Y and Z dimension kernels. We start from the Y and Z dimension FFT decompositions since the size handled by an efficient kernel is relatively smaller than their X dimension counterparts. We view the X dimension radix-FFTs as flexible helpers. Depending on the architecture influence, on the one hand, we may want to insert X dimension radix-FFTs into the decomposed Y and Z kernels as in most cases for the Tesla architecture GPUs; on the other hand, we may pad extra Y and/or Z dimension radix FFTs into the X dimension kernels as in some cases for the Fermi architecture GPUs. The bottle line is we would like a minimum number of, yet efficient, kernels with good memory-computation data dependency scheduling.

To fix the ideas, let us assume that the three-dimensional grid is of size nmp such that each of n, m, and p is large enough that it needs to be decomposed to be computed di-

rectly in registers. We call these decomposed, directly computable size k FFT as $radix-k$ FFT. The complete FFT computation needs more than one $radix - k$ FFT calls with data exchange between two calls. Such data exchange can be done through the shared memory within the same kernel or through the global memory between two kernels. Depending on the shared memory size, we may arrange $radix - k$ FFTs from the same dimension in the same or different kernels to achieve an overall minimum number of efficient kernels. This results in a thread-block configuration of the radix FFT work in the X dimension or the Y/Z dimension.

Due to the above reason, for the Tesla architecture, X dimensional $radix - k$ FFT are normally decomposed into small radixes, say $n = n_1 n_2 n_3$, while the decompositions for the other two dimensions are relatively large, say $m = m_1 m_2$ and $p = p_1 p_2$. Such a decomposition allows we complete X dimension FFT along we compute the Y and Z dimension FFT in kernels mainly for Y and Z dimensions while guarantee the efficiency of these minimum number of kernels. For most of the problem sizes in our implementations (limited by the device memory size), 4 efficient kernels would do the work.

On the other hand, for the Fermi architecture, the Y and/or Z dimension FFT size that can be handled in one single yet efficient kernel is larger than the Tesla architecture. For most of the problem sizes of our interest (limited by the device memory size), 3D forward or inverse FFT can be done in 3 efficient kernels, one for each dimension. For relatively large Y and Z dimension size, we delegate a small amount of radix FFT computations to the accommodating X dimensional kernels to guarantee the good efficiency of the Y and Z dimensional kernels. In general, we were able to save one kernel comparing to its Tesla architecture based counterpart yet all of the three kernels are of decent

efficiency.

A straightforward implementation of the basic approach will require loading and writing the 3D data for each radix computation from and into the device memory; the same applies for the scaling step. Hence the total number of the 3D data accesses from the global memory is 14, and such an implementation will not necessarily guarantee coalesced memory accesses. Our implementation attempts to achieve the following goals.

- Minimization of the total number of 3D data accesses while guaranteeing coalesced global memory access for each read and write transaction.
- All the computations are carried out on the contents of the registers directly.
- For the Tesla architecture, overlapping the small-radix FFTs along the X dimension with the FFT computations along the Y and Z dimensions, using conflict-free shared memory transposition. For the Fermi architecture, one or more rows of FFTs in the X dimension is usually computed by one kernel, with occasional extra radix computations from the Y and/or Z dimension. The majority workload, if not all of, the Y or Z dimensional FFT is computed using separate kernels using conflict-free shared memory transposition between $radix - k$ computations.
- The scaling operation should be embedded within the FFT computations. In particular, our scheme carries out the scaling operations within the last FFT computation and the first radix computation of the inverse FFT computation, thereby completely avoiding an additional 3D data access.

3.1.2 Fermi Architecture Implementation Details

We first describe the detailed algorithm for the case that different dimensional FFTs are computed in separate kernels of $n = n_1 n_2$, $m = m_1 m_2$, and $p = p_1 p_2$. We still make use of the $256 \times 256 \times 256$ size as the example. Then we extend this algorithm to accommodate problem sizes that have larger Y and/or Z dimension FFTs in which case size $512 \times 512 \times 512$ FFT execution plan is illustrated.

Each of the steps below corresponds to a kernel involving a coalesced scan of the entire 3D data.

1. X dimension FFT: thread block is configured as ($tidx$, $tidy$, 1). Upon global memory load and store, $tidy$ is for the independent X dimensional FFT and $tidx$ is for coalesced memory accesses for consecutive threads. 16 single precision complex elements are allocated to each thread. This can be used for one radix-16 FFT, or 2 radix-8 FFT, 4 radix-4 FFT,or 8 radix-2 FFT. Take X dimension 256 FFT as an example: thread block can be declared as (16, $tidy$, 1). The choice of $tidy$ can affect the occupancy by the required shared memory size and thread block size and affect the shared memory transposition pattern and padding, though 128 threads per block is typically a good choice. 16 threads compute 256 FFT as follows: 1)16 consecutive threads load 16 consecutive threads for 16 times with coalesced memory access each time; 2) compute the radix-16 FFT and multiply twiddle factors; 3) exchange data using the shared memory with bank conflict free transposition; 4) compute the second radix-16 FFT; 5) exchange data using the shared memory for coalesced global memory storing; 6) store data using a symmetric way as the first

step.

2. Y and Z dimension FFTs: thread block is configured as ($tidx$, $tidy$, 1). $tidx$ number of Y and Z dimensional (radix) FFTs are computed at once as the necessity for coalesced global memory access. Still take the 256 FFT as an example, $tidy$ is allocated to be 16 so that two rounds of radix-16 FFTs can complete the computation in that dimension as the intermediate transposition takes time and we would like to minimize such overhead as much as possible. In such a case, $tidx$ size strongly affects the device memory transaction size and 128B size is preferable to Fermi architecture since this would not waste device memory bandwidth since all the device memory requests are 128B in such scenario. On the other hand, increasing $tidx$ would increase the shared memory size per block proportionally and limit the number of resident blocks per block and as a result may stall the SM upon block synchronization. When the Y and/or Z dimension FFT sizes are too large, trying to use one kernel for the entire FFT computations in that dimension would result in prohibitively low device memory throughput and in such case, a experimentally proven effective decomposition and delegation approach would be illustrate shortly.

3. Scaling: the intermediate division scaling can be done between the last kernel of the forward FFT and the first kernel of the inverse FFT; and more these three steps can be combined into one single kernel so as to reduce global memory access rounds.

3.2 Optimized GPU Implementation for the Case of the Two-Periodic One-Neumann BC

We first discuss the special type of tridiagonal linear systems which arises in FFT-based direct Poisson solvers for this case, followed by a description of an optimized GPU algorithm.

3.2.1 Special Tridiagonal Systems

For the two-periodic one-Neumann boundary conditions, the FFT-based direct Poisson solver involves a special type of tridiagonal linear system of equations whose coefficient matrix is given by:

$$\begin{pmatrix} -1 & 1 & 0 & 0 & \cdots & 0 \\ 1 & cc[l,m] & 1 & 0 & \cdots & 0 \\ 0 & 1 & cc[l,m] & 1 & \cdots & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & \cdots & 1 & cc[l,m] & 1 \\ 0 & 0 & \cdots & 0 & 1 & -1 \end{pmatrix}$$

where $cc[l,m] = \Delta z^2 c[l,m]$ and vector d is

$$\begin{pmatrix} 0 \\ \Delta z^2 \hat{f}_{l,m,1} \\ \Delta z^2 \hat{f}_{l,m,2} \\ \vdots \\ \Delta z^2 \hat{f}_{l,m,K-1} \\ \Delta z^2 \hat{f}_{l,m,K-2} \\ 0 \end{pmatrix}$$

That is, unlike the general tridiagonal linear system introduced earlier, our application allows us to generate the coefficient matrices on the fly, and each such system involves only a few different numbers. Therefore no global memory accesses will be needed to generate the linear systems.

When Thomas' algorithm is applied to our simplified tridiagonal system, we use the variable g to store the input (that is, vector d in the initial description of Thomas' algorithm) and the output (vector x in the previous section). Since each step in the forward sweep needs the immediately preceding computed element, we use the variable g_{prev} to hold the updated g value from the previous loop to avoid loading it from the global memory. We now turn to the description of an optimized GPU algorithm. It turns out that the algorithm depends on the dimension along which the Neumann boundary conditions hold. We will describe the case when the Neumann BC is along the Z axis; the other cases can be dealt with in a similar fashion.

To simplify the presentation, we denote the size of our grid by $m \times n \times p$, where p is the size along the Z dimension. Recall that the algorithm involves p 2D forward FFTs, one for each (X,Y) slice, followed by applying the two stages of Thomas' algorithm nm times once for each index (i, j), and finally p inverse 2D FFTs are applied to generate the final output.

The GPU algorithm is straightforward: similar to the three periodic case except that the forward sweep of Thomas' algorithm is carried out in a similar as the scaling step in the previous algorithm and the backward process is carried out during the inverse FFT computations. However, there are several things to be taken care of some of the variables needed to be stored in the shared memory due to register pressure, especially for Fermi GPUs of double precision division version. Thread divergence needs to be minimized for the boundary conditions. In addition, for the double precision division, double precision intrinsic functions can be used for Fermi GPUs for efficency. The main steps are described next.

- Compute all the FFTs along the Y dimension. As before the data is initially loaded from global memory in vector format along the X dimension to ensure coalesced global memory accesses, followed by applying the in-order DIF FFT algorithm and writing back into global memory the computed values. For large values of m (size of the Y transform), we may decompose $m = m_1 m_2$ and compute radix m_1 and radix m_2 FFTs according to the shared memory size of the target GPU architecture, similar as the Y dimensional FFT in the 3D FFT.

- Compute the FFTs along the X dimension while simultaneously applying the for-

ward process of Thomas' algorithm. We load data from the global memory for the X dimensional forward FFT such that a thread block takes care of a slice of mp elements in the Y dimension. That is, each thread block computes a set of size m FFT, iterating along the Z dimension until all the FFTs in the corresponding slice are computed. Such a traversal provides a way to simultaneously implement the forward sweep of Thomas' algorithm. More specifically, threads from a block load one row of the elements in the X dimension, followed by applying the forward in-order DIF FFT algorithm to each row.Then, a shared memory transposition is performed to reconstruct the layout of the elements as they appeared right after the initial loading step. Such memory layout makes it easy to solve the tridiagonal linear system and guarantees coalesced global memory storing of the results. In the forward substitution step, we store the updated $b\,[i]$ and $v\,[i]$ into the global memory and update the $b\,[i-1]$ and $v\,[i-1]$ using the newly computed $b\,[i]$ and $v\,[i]$.

- Apply the backward process of Thomas' algorithm along the Z dimension while simultaneously computing the inverse FFTs of the vectors along the X dimension. Each thread block is responsible for loading mp elements of an (X,Z) slice. We start by loading the last row of the data, followed by applying the Z dimensional backward substitution of the last rows, store the substituted row into the "working memory" and then perform an IFFT on the last row in the X dimension, and finally storing the last row back into the global memory. Then proceed by loading the second last row from the global memory, followed by applying the backward substitution using the data stored "working memory", and so on until we are com-

pletely done with required processing along the Z and X dimensions.

- Compute the inverse FFTs for all the Y vectors in a similar way as the first step.

3.3 Performance Evaluation

The performance of our algorithms is evaluated on two Tesla architecture based GPUs: Tesla C1060 and GeForce GTX280 and two Fermi architecture based GPUs: Tesla C2050 and GeForce GTX480. The detailed specifications are listed in [72].

We compute the GFLOPS as before and compare the performance of our standalone solvers with those using CUDA FFT library for the 3 periodic BC case and the hybrid implementations with 2D CUFFT and our optimized Z dimensional tridiagonal solver for the 2 periodic 1 Neumann BC case. The specific formula used the in the evaluation can be found in [72] .

3.3.1 The Case of the Three Periodic BC

Recall that the algorithm for the three-periodic case consists of a forward 3D FFT, a scaling step, followed by an inverse 3D FFT. Hence we compare the performance of our algorithm against a GPU implementation using the CUFFT library, including or excluding the scaling step. When the scaling step is included in library-based implementations, we use a fairly optimized implementation of the 3D scaling operation and only the plan execution time is counted toward the total runtime.

We start by comparing the performance of our algorithm to the CUFFT library implementations on the Tesla C1060 and the GeForce GTX280. Figure 3.2 and Figure

3.4 show the GFLOPS performance of these algorithms on various data sizes. As can be seen from Figure 3.2, our algorithm consistently achieves around 140 GFLOPS for all the input sizes considered, much better than the CUFFT library was able to achieve even without the scaling step. Similarly, our algorithm achieves a performance ranging from 140 GFLOPS to over 160 GFLOPS on the GeForce GTX 280, again significantly better than what the CUFFT was able to achieve on the same GPU.

On the Fermi GPUs, consistently good performance were able to be achieved on both Tesla C2050 and GeForce GTX 480 as the runtime scalability shown in Figure 3.1. On the Tesla C2050, Figure 3.3, it scores between 230 GFlOPS and 290 GFlOPS for most cases and is consistently 70 GFlOPS better than the CUFFT based solver for all data sizes. Especially for the larger data sizes, CUFFT was not able to perform well and runs sharply slower. On the contrary, due to our effective technique to decompose the larger size Y and/or Z dimension FFT into smaller radix FFT and optimized radix-256 FFT, the runtime is as good as the smaller sizes. And due to the smaller radix FFT computations included in the kernels mainly for the X dimension FFT, higher GFLOPS numbers were able to be achieved. On the GeForce GTX 480, a similar good performance is demonstrated in Figure 3.5: performance ranges from 330 GFLOPS to 375 GFLOPS for these data sizes fit into the global memory and outperforms the CUFFT based counterparts by around 75 GFLOPS.

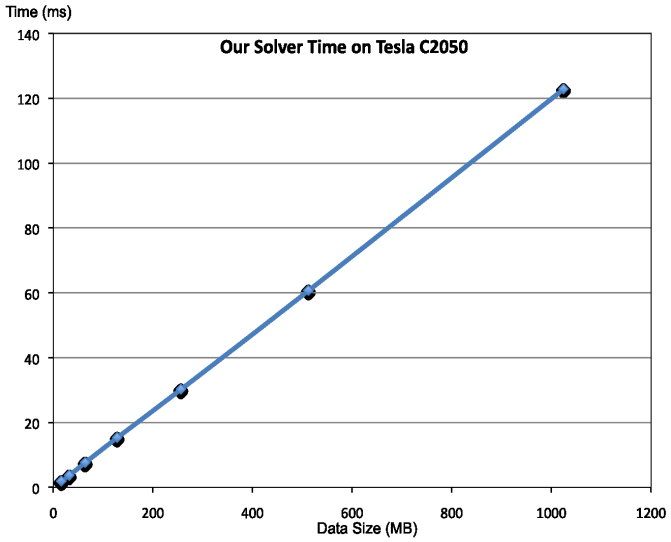


Figure 3.1: 3 Periodic BC Fermi GPUs Runtime Scalability

3.3.2 The Case of the Two-Periodic and One-Neumann BC

For the 2 periodic 1 Neumann BC case, we employ the GFLOPS performance metric as those used in the 3 periodic BC case. The number of GFLOPS consists of two components: the 2D FFT computations, and the 1D tridiagonal solvers.

We start by reporting on the performance of our algorithm on the Tesla GPUs. Figures 3.6 and 3.7 show the performance of our algorithms on the Tesla C1060 and GTX 280 on various domain sizes for single precision and double precision respectively. Due to the relatively weakness of double precision computation throughput, the Z dimensional tridiagonal solver steps can be a big burden for the X dimensional forward FFT, especially for small X dimension size. For such cases, we hybrid the highly optimized 2D CUFFT library and our optimized Z dimensional tridiagonal solver for the solver. Though this introduces two kernels with global memory accesses, each kernel could be efficient itself and performs better than the our standalone version for these cases. We capture the best

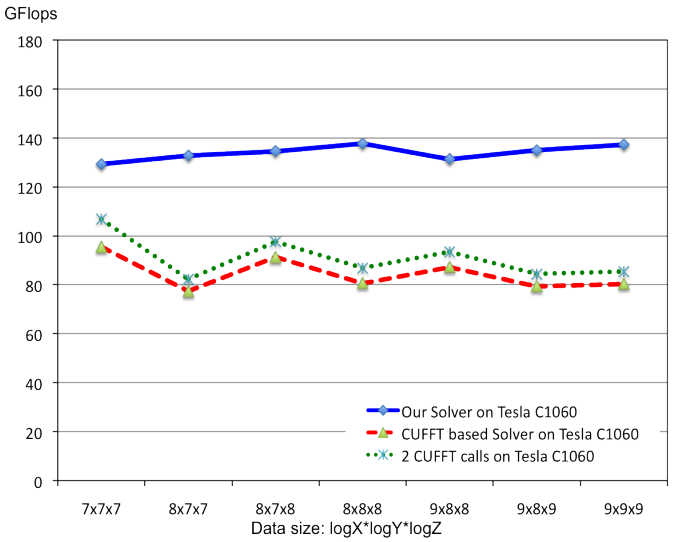


Figure 3.2: 3 Periodic Case Performance Comparison on Tesla C1060

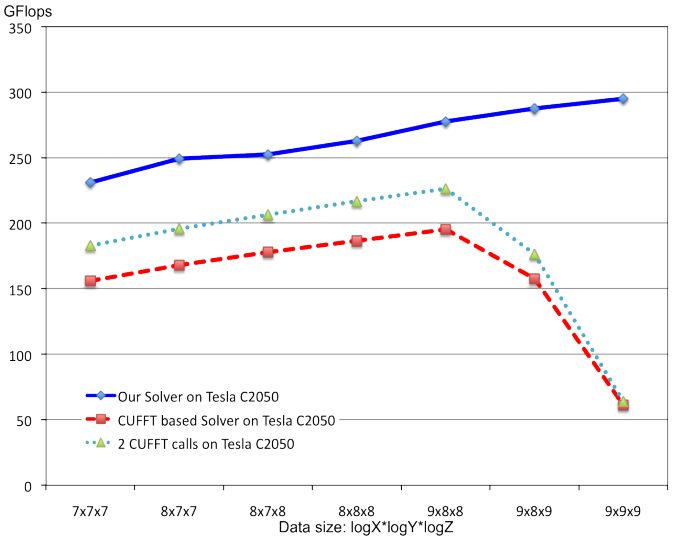


Figure 3.3: 3 Periodic Case Performance Comparison on Tesla C2050

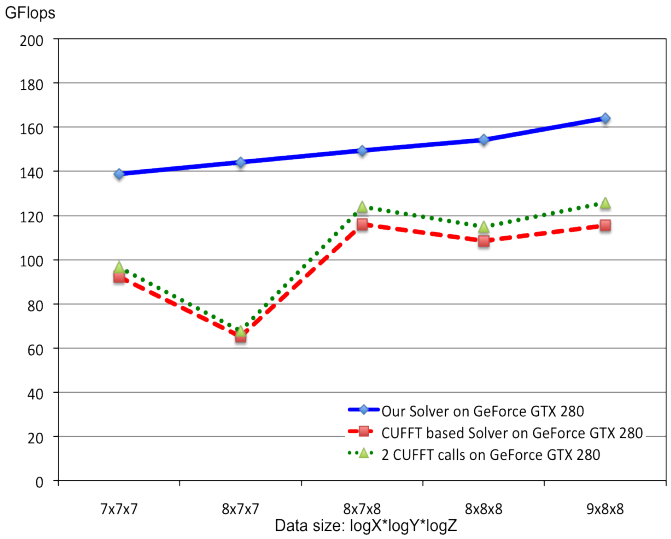


Figure 3.4: 3 Periodic Case Performance Comparison on GeForce GTX 280

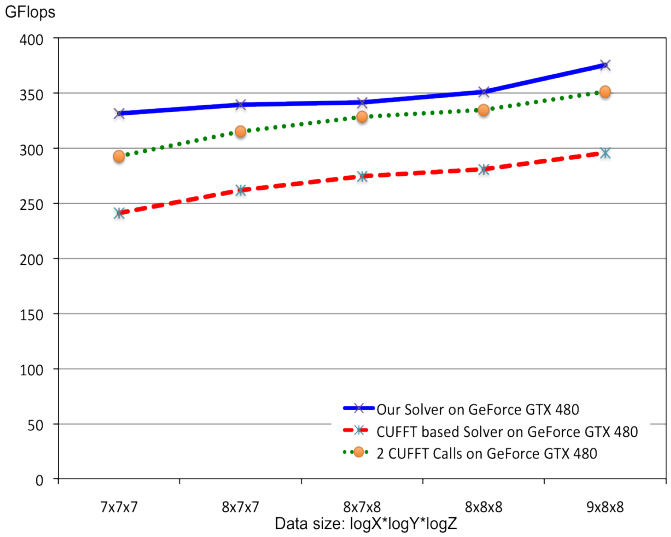


Figure 3.5: 3 Periodic Case Performance Comparison on GeForce GTX 480

GFLOPS for the two variations in our performance evaluation. Recall that the double precision indicates double precision divisor representation and double precision division in the forward reduction of Thomas' algorithm, single precision otherwise. Note the double precision GPU version enjoys the second-order accuracy while only using double precision for the division step rather than for the entire algorithm; single precision version performs slightly worse in terms of accuracy but with around 25 percent overall runtime performance gain. As indicated in these figures, the single precision solver overall performance is from 115 GFLOPS to 140 GFLOPS on the Tesla C1060 and varies from around 120 GFLOPS to 150 GFLOPS on the GeForce GTX 280. For the double precision version, the performance is around 90 GFlOPS to 110 GFLOPS on the Tesla C1060 and 110 GFLOPS to 120 GFLOPS on the GeForce GTX 280 respectively.

For the Fermi GPUs, the performance is significantly good. For single precision version (Figure 3.8), GeForce GTX 480 yields around 200 GFLOPS to 275 GFLOPS and the performance of Tesla C2050 ranges from 140 GFLOPS to 200 GFLOPS. These figures use nvcc 4.2.9; however, the performance of single precision version of using nvcc 3.2.16 is generally better than using nvcc 4.2.9, though older. The double precision version, on the other hand, degraded slightly from the single precision version, thanks to the double precision intrinsic function support (Figure 3.9). On the GeForce GTX 480, the achieved performance was between 180 GFLOPS and 220 GFLOPS and on the Tesla C2050, the achieved performance was between 130 GFLOPS and 180 GFLOPS.

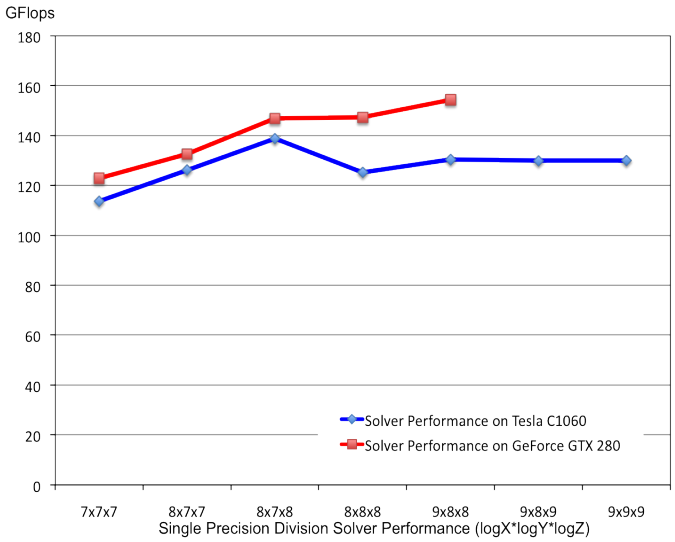


Figure 3.6: Performance of 2 Periodic 1 Neumann BC on the Tesla based GPUs (I)

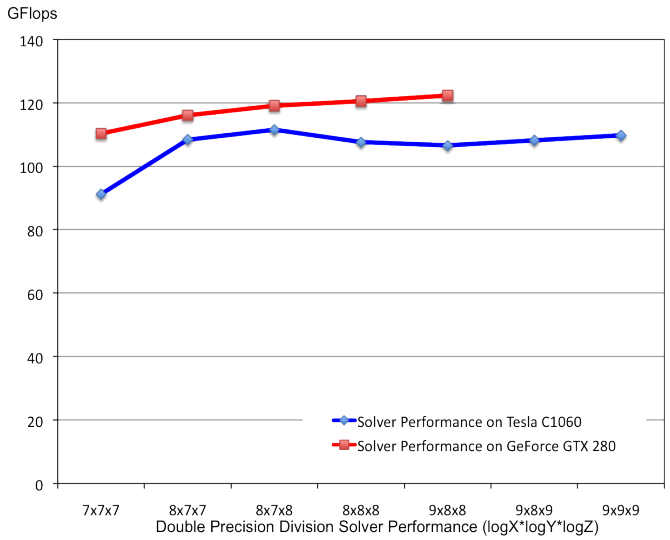


Figure 3.7: Performance of 2 Periodic 1 Neumann BC on the Tesla based GPUs (II)

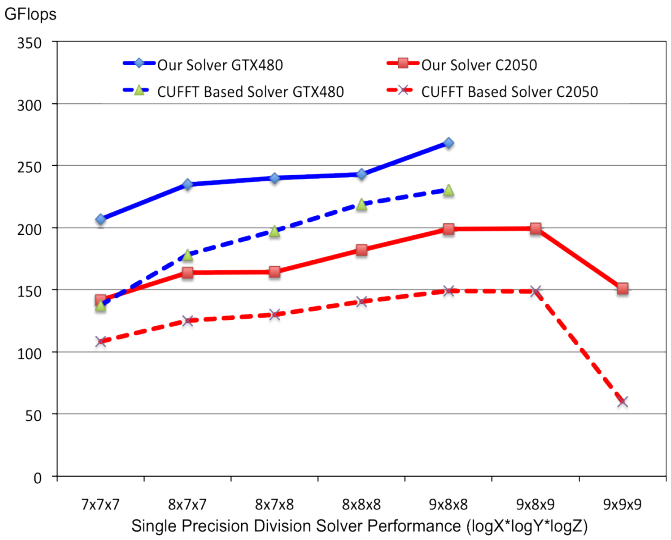


Figure 3.8: Performance of 2 Periodic 1 Neumann BC on the Fermi based GPUs (I)

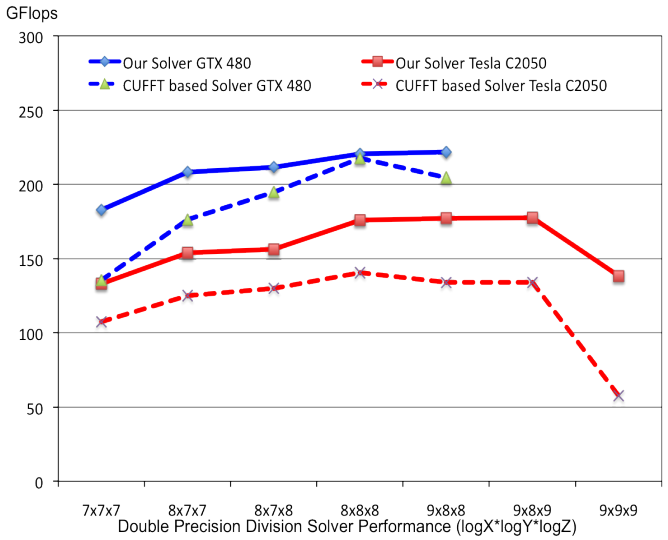


Figure 3.9: Performance of 2 Periodic 1 Neumann BC on the Fermi based GPUs (II)

Chapter 4: Out of Card Poisson Solver

We develop optimized multi-dimensional FFT implementations on CPU-GPU heterogeneous platforms for the case when the input is too large to fit on the GPU global memory, and use the resulting techniques to develop a fast Poisson solver. The solver involves memory bound computations for which the large 3D data may have to be transferred over the PCIe bus several times during the computation. We develop a new strategy to decompose and allocate the computation between the GPU and the CPU such that the 3D data is transferred *only once* to the device memory, and the executions of the GPU kernels are almost completely overlapped with the PCI data transfer. We were able to achieve significantly better performance than what has been reported in previous related work, including over 145 GFLOPS for the three periodic boundary conditions (single precision version), and over 105 GFLOPS for the two periodic, one Neumann boundary conditions (single precision version). The effective bidirectional PCIe bus bandwidth achieved is 9-10GB/s, which is close to the best possible on our platform. For all the cases tested, the single 3D data PCIe transfer time, which constitutes a lower bound on what is possible on our platform, takes almost 70% of the total execution time of the Poisson solver.

4.1 Introduction

There has been recent interest in the development of high performance direct Poisson solvers due partly to the introduction of immersed-boundary methods [64]. A Poisson solver is an extremely important tool used in many applications, which most often constitute the most computationally demanding component of the application. In an earlier work [70], we developed an FFT-based direct Poisson solver for GPUs, which was optimized for the case when the 3D grid fits onto the device memory. The performance reported there assumes that both the input and output reside on the device memory, which is the typical assumption made by most of the published GPU algorithms. In this chapter, we consider the case when the grid is much larger than the size of the device memory, but can still fit in the main memory of a host multicore CPU, and develop optimized FFT computations, and FFT-based direct Poisson solver on such platforms, which significantly expands our earlier work in [73]. Our approach [74] exploits the particular strengths of each processor while carefully managing the data transfers needed between the CPU and the GPU. In particular, our algorithm includes optimized 2D or 3D FFT implementations and optimized tridiagonal solver implementations for such heterogeneous environments in which both the input and the output reside in the main memory of the CPU.

Most of the recently published work of FFT algorithms on GPUs [68, 75–79], assume data sizes limited by the device memory size. This assumption results in efforts that are concentrated on GPU optimization, including data transfers between device memory and the shared memory or registers of the streaming multiprocessors. For memory bound computations, such as FFTs, the performance bottleneck becomes the device memory

bandwidth and the type of the global memory accesses. For recent GPUs, the peak device memory bandwidth can be 100-200 GB/s.

We compare our results to two recent results on a similar model. Chen et al [48] used a cluster of 4 or 16 nodes, each node includes two GPUs (Tesla C1060 and GTX 285), to handle large 3D FFT computations. They reported a performance of around 50 GFLOPS on four nodes, somewhat lower than our performance on a single node with a Tesla C1060 (in fact, our performance number is an under-estimate since it does not take into consideration all the components of our Poisson solver). Another recent work is reported by Gu et al [80], which tries to optimize both CPU-GPU data transfer and GPU computations for 1D, 2D, and 3D FFTs. In particular, they develop a blocked buffered technique for 1D FFTs which achieves a high bandwidth on the CPU-GPU data channel. For their multidimensional FFTs, the data has to be transferred back and forth between the CPU and GPU at least twice, and for 3D double-precision FFT, their best performance is around 15 GFLOPS on the NVIDIA Tesla C2070, 13 GFLOPS on the NVIDIA GTX480 and 9 GFLOPS on the NVIDIA Tesla C1060 respectively. Our performance numbers for the single-precision FFTs reach 60 GFLOPS using the Tesla C1060. And when using Tesla K20, which supports bidirectional PCIe bus transfers (similar as Tesla C2070), we achieved more than 140 GFLOPS for the single precision FFTs and more than 70 GFLOPS for the double precision FFTs.

Our main contributions can be summarized as follows.

- The computation is organized in such a way that the 3D grid data is transferred between the CPU memory and the device memory only once, while achieving a

PCIe bus bandwidth close to the best possible on our platforms.

- The GPU kernel computations are almost completely overlapped with the data transfers on the PCIe bus, and hence the GPU execution time contributes very little to the overall execution time. This is due to an effective use of the CUDA page-locked host memory allocation, asynchronous function calls, stream scheduling, and write-combining.

- Our CPU-GPU workload decomposition is equally effective for both single precision and double precision implementations. While our single precision implementation achieves an accuracy comparable to a double precision implementation, it achieves double the GFLOPS for the same data sizes.

- Experimental tests on our platform for problems of large sizes show that almost 70% of the total execution time is consumed by the single 3D grid data transfer over the PCIe bus, and most of the rest is consumed by the initial CPU computation of the FFT along the X dimension. The overall performance of our FFT-based Poisson solver ranges of 50-60 GFLOPS for a relatively older CPU-GPU platform and around 140 GFLOPS for a newer platform.

4.2 Overview and Background

In this section, we provide an overview of the algorithms behind the FFT-based Poisson solver, which include FFT and tridiagonal linear system computations. Basic FFT algorithms that are related to our work are then summarized, followed by an overview

of Thomas' algorithm for solving tridiagonal linear systems. We end this section with an overview of the general architecture of our platforms that consists of a multicore processor with a GPU accelerator.

4.2.1 Architecture Overview

Our experimental platforms are heterogeneous processors, each of which consists of a multi-core CPU and a GPU accelerator, such that the CPU memory is substantially larger than the GPU device memory. More specifically, we use two testbeds for our work. The first is a dual socket quad-core Intel Xeon X5560 CPU with 24GB main memory and an NVIDIA Tesla C1060 with 4GB device memory - we refer to this testbed as the *Nehalem-Tesla* node, after the codename of the CPU and the architecture of the GPU respectively. The second is a dual socket octal-core Intel E5-2690 with 128GB main memory and an NVIDIA Tesla K20 with 5GB device memory - we refer to this testbed as the *Sandy-Kepler* node (we use Sandy rather than Sandy Bridge for brevity). The input data is much larger than the device memory and is assumed to be initially held in the CPU memory. At the end of the computation, the output data must reside in the CPU memory as well. Data transfers between the CPU main memory and the GPU device memory are carried out by a PCIe 2.0 bus: unidirectional for the *Nehalem-Tesla* node (compute capability 1.3) and bidirectional for the *Sandy-Kepler* node (compute capability 3.5).

4.2.1.1 CUDA Programming Model

The CUDA programming model assumes a system consisting of a host CPU and a massively data parallel GPU acting as a co-processor, each with its own separate memory [81]. The GPUs consist of a number of Streaming Multiprocessors(SMs), each of which containing a number of Streaming Processors (SPs or cores). The GPU executes data parallel functions called kernels using thousands of threads. The mapping of threads onto the GPU cores are abstracted from the programmers through - 1) a hierarchy of thread groups, 2) shared memories, and 3) barrier synchronization. Such abstraction provides fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism and this is based on similar hardware architecture among generations. Details of the CUDA programming model can be found at [81] and we will only refer to the aspects that are key to our optimization scheme. In this work, we are concerned with Tesla C1060 and K20 whose main features are summarized in Table 5.1. Note that, for the Tesla K20, the L1 cache and the shared memory per SM share a total amount of 64KB on-chip memory whose ratio is configurable.

4.2.1.2 PCIe bus

The PCIe 2.0 bus between the CPU and GPU is of central importance for large size problems and for memory bound computations such as ours. The PCIe 2.0 has a theoretical single directional peak bandwidth of 8 GB/s - with a relatively smaller best achievable bandwidth in our evaluation. On the *Sandy-Kepler* node, single directional memory transfer from pinned host memory to device memory (H2D-host to device) reaches

Table 4.1: The Two GPUs Used in This Chapter

GPUs	Tesla C1060	Tesla K20
SMs per GPU	30	14
SPs per SM	8	192
Registers per SM	16K	64K
Shared Mem per SM	16KB	16KB-48KB
L1 Cache per SM	NA	48KB-16KB
L2 Cache per GPU	NA	1.25MB
Global Mem per GPU	4GB	5GB
GPU clock rate	1296MHz	706MHz
Memory clock rate	800MHz	2600MHz
Memory bandwidth	102.4GB/s	208GB/s
Compute Capability	1.3	3.5

5.7GB/s bandwidth and from device memory to pinned host memory (D2H-device to host) achieves 6.2GB/s bandwidth. However, when bidirectional memory transfer is done concurrently, a further slight bandwidth degradation is observed: 5.44GB/s for D2H and 5.34GB/s for H2D are the best we were able to achieve for pure bidirectional memory transfer with varying data sizes. This gives a combined 10.78GB/s upper bound on the best achievable bandwidth. On the *Nehalem-Tesla* node, only single directional memory transfer is supported and the observed H2D bandwidth is 5.4GB/s and for a D2H copy the best bandwidth achievable is 5.3GB/s. Similar observations were reported by others including NVIDIA [80, 82]. Clearly the data transfer between the host and the device memories constitutes the major bottleneck for our problem.

4.2.1.3 Multi-core CPU

In addition to acting as the CUDA host, the multicore CPU offers in itself a multi-threaded environment with a shared memory programming model. In most previous work,

the focus has been on GPU optimization without trying to make use of the CPU computational resources. In our approach, we make use of the multicore CPU in two ways: 1) we allocate part of the computation to the CPU cores and partition the CPU and GPU work in such a way that the GPU work requires only one iteration of data transfer over the PCIe bus; 2) we use the multi-core CPU to enable concurrent asynchronous transfers between the host memory and the pinned memory: unidirectional for the *Nehalem-Tesla* node and bidirectional for the *Sandy-Kepler* node. In addition, modern multi-core CPUs are built with SIMD support: SSE is supported on the Xeon X5560 and AVX is supported on the Xeon E5-2690. Such features allow us to carry out a limited amount of data intensive parallel computations quite effectively on the CPU.

4.2.1.4 Asynchronous CUDA streams

CUDA supports asynchronous concurrent execution between host and device through some asynchronous function calls - control is returned to the host thread before the device has completed the requested task [81]. Data transfer and kernel execution from different CUDA streams [81] can be overlapped when memory copies are performed between page-locked host memory and device memory. Some devices of compute capability of 2.x and higher (K20 in our evaluation) can perform memory copy from page-locked host memory to device memory (H2D) concurrently with a copy from device memory to page-locked host memory (D2H). With careful orchestration of the CPU work and CUDA streams, we essentially establish a CPU-GPU work pipeline of depth of four (for the *Nehalem-Tesla* node) and five (for the *Sandy-Kepler* node) in which computation and communication

are almost completely overlapped. Moreover, our effective CPU-GPU work pipeline of bidirectional PCIe bus transfer essentially doubles the PCIe bus performance of the uni-directional PCIe bus transfer version.

4.3 Overall Approach

In this section and the following section, we describe our overall strategy to handle the FFT-based direct Poisson solver computations for the cases of three periodic boundary conditions, and the two periodic, one Neumann boundary conditions. In each case, we describe how the overall computation is decomposed and scheduled onto each of the CPU-GPU platforms, and how data transfers between the CPU memory and the GPU global memory are managed to cause an almost complete overlap between computation and data transfer.

4.3.1 Three Periodic Boundary Condition Case

The 3 periodic Boundary Condition (BC) case involves a 3D forward FFT, a scaling of each element, and a 3D inverse FFT. The scaling (division) of each element during the intermediate step depends only on the 3D indices of the element, which allows us to incorporate the scaling operations within the forward FFT or inverse FFT computations. In our implementation, we choose the in-order input FFT DIF variation for the forward FFT, and the in-order output FFT DIT variation for the inverse FFT computation. A straightforward implementation of the 3D FFT algorithm would require moving the 3D data once along each dimension, resulting in the 3D data being exchanged between the

CPU and the GPU over the PCIe bus three times.

We start by noting that the CPU cores offer opportunities for a significant amount of parallelism on highly irregular computations, and that the availability of caches makes the CPU quite effective in handling FFTs along the X dimension due to the memory layout of the 3D data. Note also that the SIMD capability of the CPU presents possibilities for additional performance enhancement. On the other hand, the GPU architecture is much more effective for massive data parallel computations using more structured memory access patterns. Therefore, we decompose the overall work among the CPU and the GPU in such a way that: (1) the volume of the data transferred over the PCIe bus is minimized. In our case, the 3D data will be transferred only once between the two devices; (2) the FFT computations along the X dimension will be effectively carried out by the CPU cores; and (3) the rest of the FFT computations will be carried out by the GPU cores through a sequence of asynchronous streams of chunks of the 3D data. Each asynchronous stream will go through a 5-stage pipeline consisting of: data transfer from the host system memory to the host pinned memory; memory copy from the host to the device memory (H2D); GPU kernel executions; memory copy from the device to the host pinned memory (D2H); and data transfer to the host system memory. We orchestrate the data movements to overlap H2D memory copy, kernel execution, and D2H memory copy.

We illustrate our strategy in details by focusing on the problem of size $1024 \times 1024 \times 1024$. Similar strategies work as effectively for other sizes.

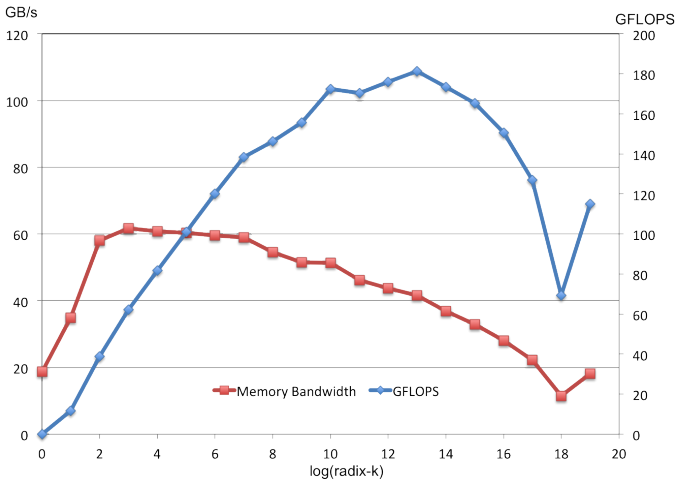


Figure 4.1: Performance of Batched 1D DFT Using MKL library

4.3.1.1 Multi-threaded CPU forward X dimensional FFT

As mentioned before, the FFT computations along the X dimension are carried out by the CPU cores. We make use of Intel's Math Kernel Library (MKL) SIMD OpenMP based DFT routines to execute this step. This library seems to effectively exploit the multicore architecture, the memory hierarchy, and the SIMD capability of the core processors. As an example, we demonstrate the performance of this library on batches of one-dimensional FFTs of sizes ranging from 2^0 up to 2^{19} on the CPU of the *Sandy-Kepler* node. The results are shown in Figure 4.1, where the performance is illustrated through two curves - one showing the GFLOPS performance and the second showing the memory bandwidth achieved as a function of the input size assuming only one memory read and store were done for each element. As can be seen from this figure, the memory bandwidth achieved is quite good (relative to the peak of 79.55GB/s reported in [83]), especially in the range we are interested in (between 1K and 4K). While the GFLOPS performance varies over a relatively significant range, It is quite good over the range of interest to us

(between 1K and 4K). Therefore, the forward and the inverse FFTs along the X dimension are completed by calling the MKL library.

4.3.1.2 Asynchronous Streams of Data Movements and GPU Kernels

CUDA allows the use of streams for asynchronous memory copy and concurrent kernel executions to hide long PCIe bus latency [81]. A stream is a sequence of commands that execute in order; different streams may execute their commands out of order with respect to one another or concurrently. Asynchronous memory copy has to be carried out between page-locked host memory and device memory. The H2D and D2H memory copies can be done concurrently on the Kepler GPUs but only one-directional memory copy can be executed at a time on the Tesla GPUs. This would result in a slightly different organization of the CPU and GPU pipeline on each platform.

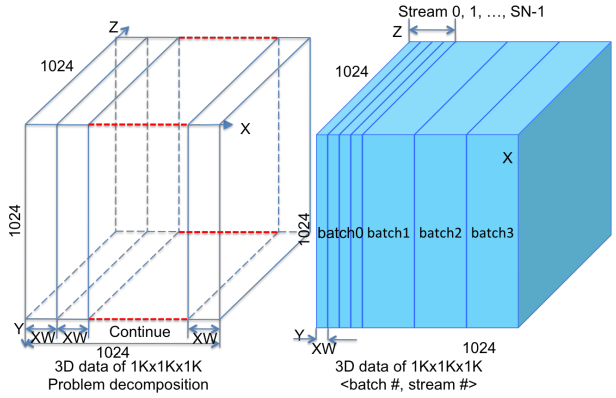


Figure 4.2: 3D data memory layout

We now focus on the *Nehalem-Tesla* node and later address the streams used for the *Sandy-Kepler* node. In order to make effective use of the asynchronous CPU-GPU memory copy for our running example, we organize the remaining FFT computations into

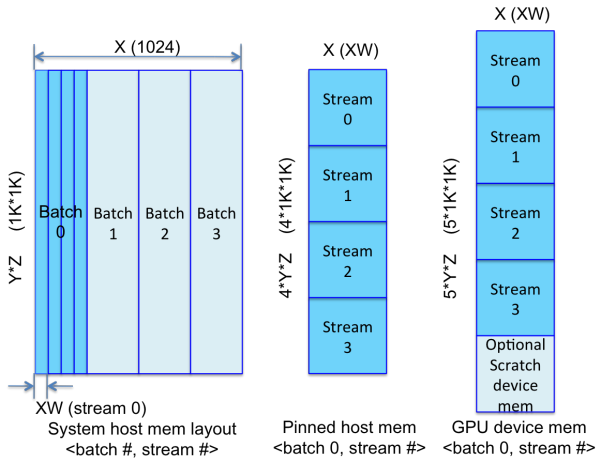


Figure 4.3: CPU and GPU device memory usage

four batches, each consisting of four asynchronous streams where each stream involves a subarray of size $64 \times 1024 \times 1024$ (0.5 GB) - this means a vector size of 64 along the X dimension, which is demonstrated as "XW" in Figure 4.2. The choice of vector size 64 is determined to optimize the use of the PCIe bus bandwidth. The corresponding memory layout of the problem decomposition is shown in Figure 4.2.

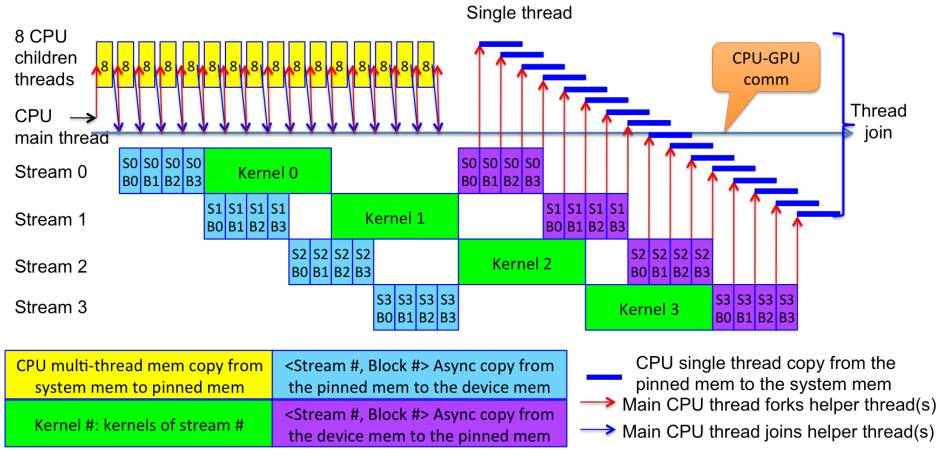


Figure 4.4: CPU-GPU Pipeline for *Nehalem-Tesla* Node

For our running example, staging page-locked host memory of size 2GB (0.5GB*4) is allocated to enable asynchronous memory copy, as indicated in Figure 4.3. By default,

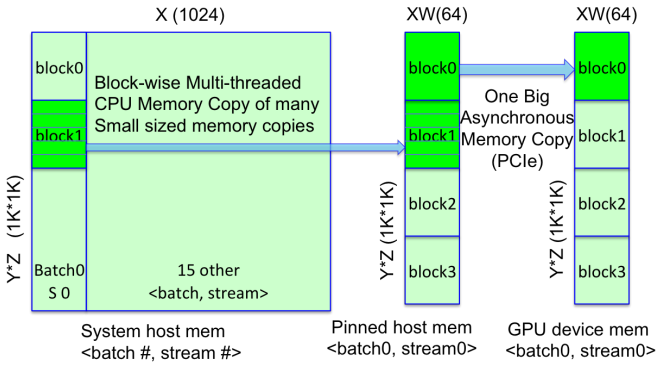


Figure 4.5: block memory copy for one stream (Tesla C1060)

page-locked host memory is allocated as cacheable and *write-combining* flag can be used to enable the memory not to be snooped at during data transfer across the PCIe bus, which can boost the host to device bandwidth in practice [81]. However, the bandwidth on the opposite transfer direction is prohibitively slow. So we allocate two scratch page-locked memories: one with default flag and using for device to host transfer and one with *write-combining* flag and using for host to device transfer.

Figure 4.4 shows one batch of the complete pipelined execution of multi-threaded CPU (including the main thread and the helper threads) and 4 GPU streams (stream 0, 1, 2, 3). Each stream is defined as follows.

- The 3D data subset allocated to each stream is $64 \times 1024 \times 1024$ along the X, Y and Z dimensions respectively. This corresponds to the system host memory layout versus <batch#, stream#>in Figure 4.5, which indicates $1K \times 1K$ lines of 64 8-byte words with 1024×8-bytes stride between every two lines. Each line is denoted by XW in the figure, corresponding to the X-dimensional-Width.

  These apart elements need to be packed consecutively in the page-locked memory so that the following PCIe bus transfer bandwidth would be effective. The data

movement for each data subset is a pipeline of block-wise movement involving a multi-threaded CPU memory copy of a large number of 64-element words into a consecutive block in the paged-locked memory, followed by a PCIe bus transfer. The data movement from the system host memory to the pinned host memory and the data movement from the pinned host memory to the device memory is simultaneous as indicated by the two arrows in Figure 4.5.

The entire process overlaps PCIe bus transfers with multi-threaded CPU data copy into pinned memory. Due to bandwidth differences of the PCIe bus and the multi-threaded system memory copy, by the time PCIe bus is done with the previous sub-chunk, the next sub-chunk will be ready for the asynchronous memory copy into the device memory.

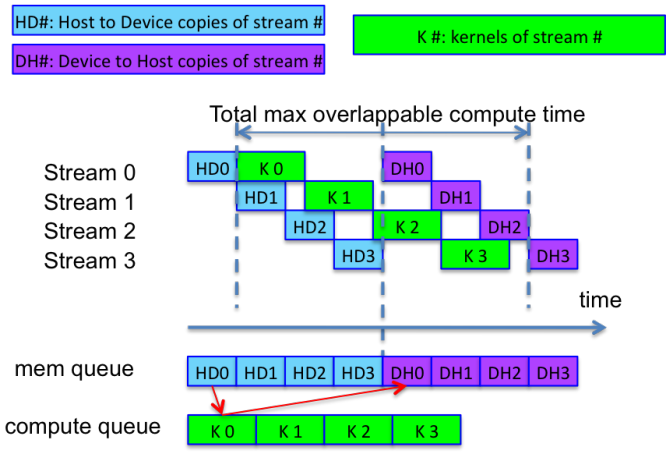


Figure 4.6: Async CUDA streams for Tesla C1060

Immediately after we execute the memory copy for one chunk of $64 \times 1024 \times 1024$ data, and launch the asynchronous kernel calls for that stream and start the same work of the next $64 \times 1024 \times 1024$ data chunk. Upon the completion of the kernel calls, we make use of asynchronous copy attached to the same stream for the copy

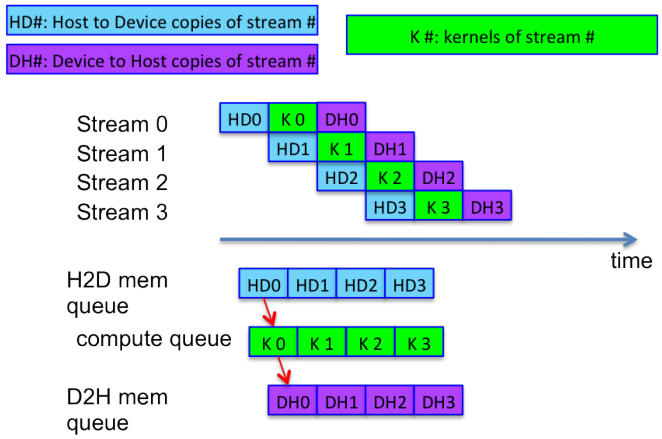


Figure 4.7: Async CUDA streams for Tesla K20

back. However, due to the limitation of Tesla C1060, there are no concurrent data transfers back and forth between pinned memory and device memory. When we schedule the asynchronous work, we have to schedule the copy back calls after executing all the copying from the pinned host memory to the device memory and their kernel calls. This asynchronous stream execution is shown in Figure 4.6.

- Compute the 2D forward FFT, scaling and 2D inverse FFT computation (of 64 along the X dimension) on a chunk of size $64 \times 1024 \times 1024$ on the GPU using 7 optimized kernels. The total execution time of the kernels (of the 4 streams) should be smaller than the total transfer time of 3 streams (3 host to device and 3 device to host, Figure 4.6); otherwise, one or more of the streams' memory transfer back needs to be held back until the completion of its kernel. This is illustrated in Figure 4.6. Since we want to achieve a high PCIe bus bandwidth, the kernels have to execute as fast as well. Once the data is loaded onto the GPU device memory, we can use techniques similar to those introduced in our previous work [70] to compute the Y and Z dimensional FFTs of each subarray of size $64 \times 1024 \times 1024$. An

intermediate global memory (shared by 4 streams due to their sequential execution of kernels) is introduced for smaller strides between consecutive global memory accesses when multiple Z dimensional computation kernels are involved (Figure 4.3), without limiting the maximum number of concurrent streams. The scaling step is included in the last step of the forward FFT and the first step of the inverse FFT with the scalars computed using bit-reversed indices.

We borrow the following notation from our earlier work [70]: $\{Y(p, q, r, n), forward\}$ amounts to the execution of p radix-q forward FFTs along the Y dimension with a stride of r with a group size of n. Using this notation, the GPU kernels can be defined by the following computations:

- $\{Y(32, 32, 32, 1024), forward\}$
- $\{Y(32, 32, 1, 32), forward)\}$
- $\{Z(32, 32, 32, 1024), forward\}$
- $\{Z(32, 32, 1, 32), forward\}$

  $\{scaling, GPU)\}$

  $\{Z(32, 32, 1, 32), inverse\}$
- $\{Z(32, 32, 32, 1024), inverse\}$
- $\{Y(32, 32, 1, 32), inverse\}$
- $\{Y(32, 32, 32, 1024), inverse\}$

Note that all the arithmetic computations are carried out on register contents, all global memory transfers involve coalesced memory access (the vector size along the X dimension is selected to ensure the global memory coalescing). Therefore,

we complete the 64 sets of 1024×1024 forward FFT, scaling and inverse FFT using 7 kernels.

- Once the kernels are completed, we perform block-wise asynchronous memory copy from the device memory to the pinned host memory and then to the system host memory for each stream. *cudaStreamSynchronize()* is used to let the CPU memory copy back wait for the completion of the asynchronous GPU-to-CPU memory copy for that data chunk (Figure 4.4).

4.3.1.3 Asynchronous Streams of Data Transfers and GPU Kernels for the *Sandy-Kepler* Node

On the *Sandy-Kepler* node, memory transfers between the host memory and the device memory are possible in both directions concurrently. Therefore, rather than postpone the memory transfer of the next batch from the host memory to device memory until the completion of device memory to host memory transfer, the next batch of memory transfer could start immediately as long as the pinned host memory portion used by the same stream in the previous batch is copied into the device memory, namely, we want to ensure no overwrite hazard is possible as illustrated in Figure 4.8. Only in this way bidirectional memory transfer could be maintained between batches without the pipeline being underfed - essentially we need to establish a 5-stage pipeline: 1) S2P memory copy; 2) H2D memory copy; 3) kernel execution; 4) D2H memory copy; and 5) P2S memory copy. This implies that we need at least 5 streams of data movements and GPU computations for a non-stalling pipeline.

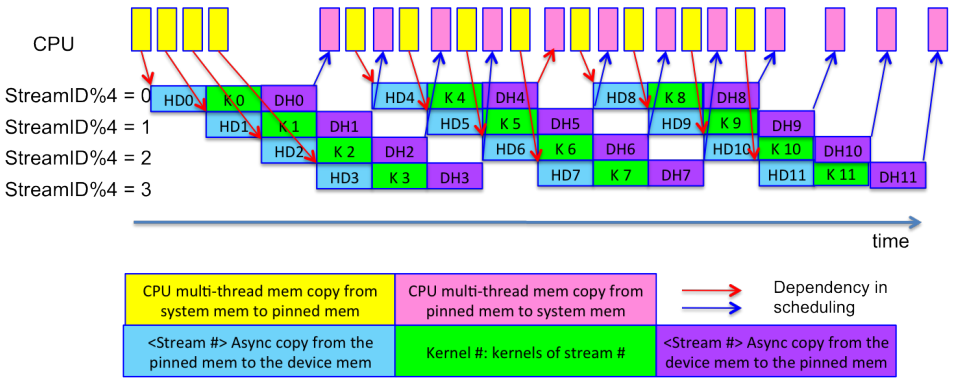


Figure 4.8: CPU-GPU Pipeline for *Sandy-Kepler* Node

Moreover, CUDA 5.0 added a new runtime function that allows the insertion of a callback function at any point in a stream. Such a callback function is executed on the host once all commands issued on the stream before the callback have been completed. We employ the callback for the data movement between the pinned host memory and the system host memory for that stream. The callback function for each data subset needs a private memory space to store the information about the source and destination addresses of the data subset. Because of the asynchronous execution of the memory copy and the kernel launches, we use separate space for each data subset to avoid any type of data hazards. Therefore, a straightforward implementation would be to assign each data subset to a stream, but the stream is scheduled in such a way that in the intermediate execution a fixed number of streams are active as illustrated in Figure 4.8 - four streams were illustrated in the figure for clarity.

In order to minimize the non-overlapping transfer time of the first and last streams, we try to reduce the size of each data subset (while still large enough to achieve high PCIe bandwidth). On the other hand, we need to guarantee the efficiency of the resultant GPU kernels - one key feature being to ensure coalesced global memory access. As 128-byte

being the largest device memory transaction size and the GPU L2 cache line size, we choose 128 bytes as the X dimension size of each data subset. As we already completed the X dimensional FFT, the choice of the 128 bytes for the X dimension is merely to optimize the GPU memory throughput performance during kernel execution. Last, we avoid the block-wise memory copy technique used in the *Nehalem-Tesla* node since each subset is already small and the overhead of blocking the subset could not be justified based on our tests.

For concreteness, let us focus on the single precision case for our running example. The entire 1Kx1Kx1K data set is divided into 64 sets, each of size 16x1Kx1K (128MB) and organized into 64 asynchronous streams. The double precision version is merely half of the number of elements along the X dimension for each stream. The pinned host memory is large enough to hold 8 data subsets. These 64 streams are mapped into the 8 slots, eight at a time, and scheduled into execution in a round-robin order while data hazards are avoided through a shared status update protected by a MUTEX. Specifically, the possible data overwrite can only happen between streams that map to the same pinned host memory space one after another. As we are using two pinned host memory spaces for the sake of better PCIe bus bandwidth, the forward copy pinned host memory space is available for the next stream as long as the data is copied to the GPU's device memory. That is, we can proceed after the completion of the asynchronous memory copy from the pinned host memory to the GPU's device memory. As we are already using a CUDA stream callback upon the completion of the asynchronous memory copy back from the device memory to the host memory and we have enough concurrent streams ready to feed the PCIe bus, we postpone this "Green" light status update in the callback function right

before the multi-threaded memory copy from the pinned host memory to the system host memory. Later, in the next round, the CPU main thread would check if the "Green" light is on before launching another streaming of copy data from the system host memory to the pinned host memory, otherwise, it would go to sleep for a while and repeat.

As a result, the GPU kernels can be defined by the following computations:

- $\{Y(4, 256, 4, 1024), forward)\}$
- $\{Y(256, 4, 1, 4), forward)\}$
- $\{Z(4, 256, 4, 1024), forward)\}$
- $\{Z(256, 4, 1, 4), forward)\}$

  $\{scaling, GPU)\}$

  $\{Z(256, 4, 1, 4), inverse)\}$
- $\{Z(4, 256, 4, 1024), inverse)\}$
- $\{Y(256, 4, 1, 4), inverse)\}$
- $\{Y(4, 256, 4, 1024), inverse)\}$

4.3.1.4 Multi-threaded CPU inverse radix FFT computation

This step is similar to the first step - we use the MKL library to compute the X dimensional FFT with batched execution using all available cores.

4.4 2 Periodic 1 Neumann Boundary Condition Case

4.4.1 Algorithm

Suppose our 3D input data is of size $NX \times NY \times NZ$. The 2 periodic 1 Neumann BC case involves NZ sets of 2D forward FFTs, each of size $NX \times NY$, followed by $NX \times NY$ sets of tridiagonal linear systems, each of size $NZ \times NZ$, followed by NZ sets of 2D inverse FFT of size $NX \times NY$. We use a strategy similar to the one used before to decompose the computation between the CPU and GPU while carefully organizing streams of data transfers between the two devices.

4.4.2 Strategy

We illustrate our strategy for the case of $1024 \times 1024 \times 1024$, and examine in some detail how the work is allocated between the CPU and the GPU for this case. The same strategy works for other problem sizes as we demonstrate later. We start with the specific details for the *Nehalem-Tesla* node, followed by the details for the *Sandy-Kepler* node.

4.4.2.1 Details on the *Nehalem-Tesla* Node

- As before, the first step is carried out on the CPU,using a batch of 1D X dimensional MKL FFT library calls on all the available CPU cores.
- We launch a set of asynchronous streams involving memory copy such that each of the streams performs the following computations of data size $64 \times 1024 \times 1024$ running on the GPU:

– Compute the forward FFTs along the Y dim:

$\{Y(1, 1024, 1, 1024, forward, GPU)\}$

– Using Thomas' algorithm, solve the tridiagonal linear systems of equations

$\{Z(1024, tridiagonal\ solver, GPU)\}$

– Compute the inverse FFT along the Y dim:

$\{Y(1, 1024, 1, 1024, inverse, GPU)\}$

- After the GPU completes the execution of all the kernels and the intermediate results are written back in the CPU main memory, we execute a batch of 1D X dimensional MKL inverse FFT library calls on the available cores.

Note that, once a chunk is loaded into the GPU global memory, we ensure a fast GPU execution by minimizing the number of global memory accesses, all of which are guaranteed to be coalesced.

The CUDA streams are employed to combine the CPU and GPU work using asynchronous memory copy and kernel executions in a similar way to what we did for the 3 periodic BC case: for our running example, 4 streams achieve a very good PCIe bandwidth (around 4.5GB/s) on the *Nehalem-Tesla* node.

4.4.2.2 Details on the *Sandy-Kepler* Node

On the *Sandy-Kepler* node, a similar strategy using the MKL DFT library calls is equally effective. The only difference of the 2 periodic 1 Neumann BC case from the 3 periodic BC case on the *Sandy-Kepler* node is that the Z dimensional kernels are done using different kernel functions and separate scratch space is allocated for the corresponding

data set to store vector B.

As a result, the GPU kernels can be defined by the following computations:

- $\{Y(4, 256, 4, 1024),\ forward)\}$
- $\{Y(256, 4, 1, 4),\ forward)\}$
- $\{Z\ dim\ forward\ reduction)\}$
- $\{Z\ dim\ backward\ elimination\}$
- $\{Y(256, 4, 1, 4),\ inverse)\}$
- $\{Y(4, 256, 4, 1024),\ inverse)\}$

4.4.3 Arithmetic Precision

When it comes to GPU performance, single precision floating point arithmetic enjoys significant benefits over double precision arithmetic [84]. Since single precision floating points use half of the memory space of double precision floating points, single precision implementations potentially save half of the memory transfer time, for the PCIe bus and for the global memory accesses. Also, single precision computations are faster than double precision computations on many architectures, including the two GPUs we are using. An important characteristic of our algorithm is to secure a 2^{nd} order convergence, and hence if we make the grid twice as dense, the accuracy would be four times better. In our experiments, double precision arithmetics can easily guarantee such property at the expense of slower computation time, while pure single precision implementations showed a relatively larger error when compared to the discretized analytic function used in our tests. And due to the slow PCI peak bandwidth and fast GPU kernels, these

two variations show almost the same performance in our experiments.

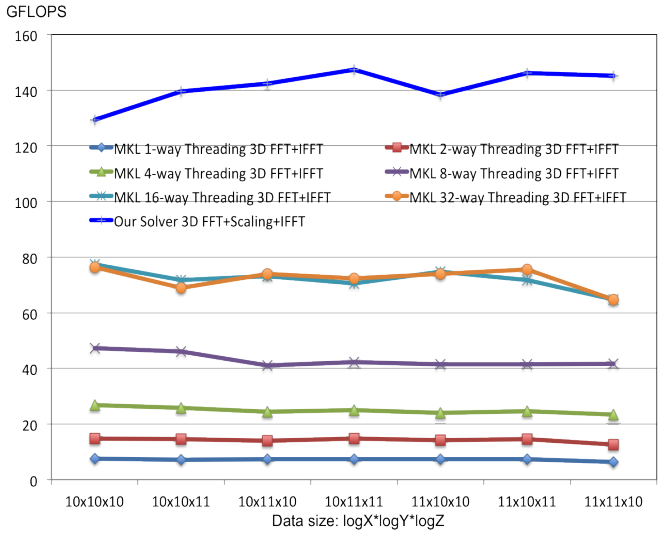


Figure 4.9: 3 Periodic BC Single Precision Perf. on the *Sandy-Kepler* Node

To achieve high performance while ensuring the 2^{nd} order convergence, we make use of a precision boost for the intermediate data. Through careful examination, we notice that the step that most affects the precision is the division step in the forward elimination stage: $m = a[i]/b[i-1]$. More specifically, the error becomes large when $b[i-1]$ is small. Note that in our implementation, the $b[i-1]$ is stored and updated as we iterate along i. Hence we use double precision to store the $b[i-1]$ values and immediately related variables, and then cast the results back into single floating points. By using this trick, we can avoid the performance degradation of converting the entire data into double precision while achieving the desired accuracy.

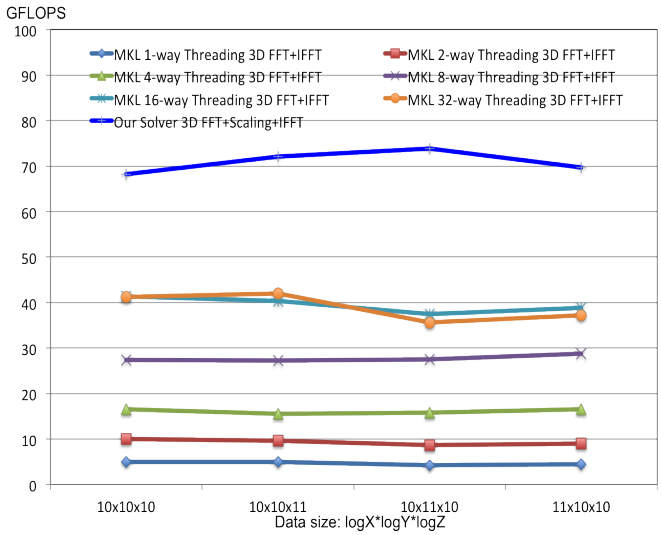


Figure 4.10: 3 Periodic BC Double Precision Perf. on the *Sandy-Kepler* Node

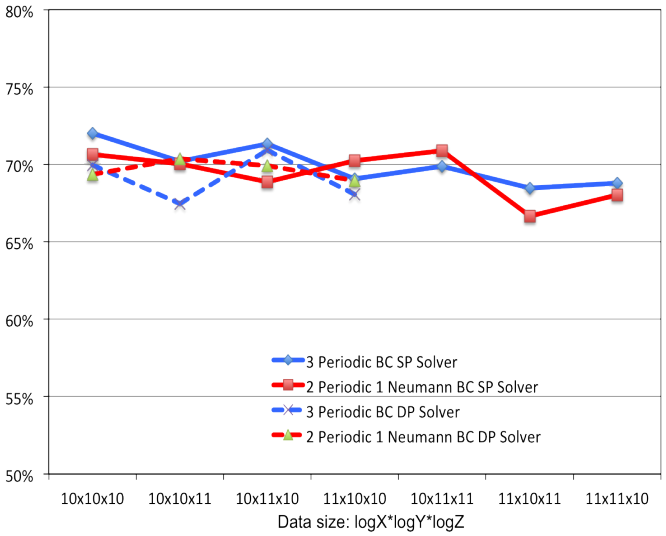


Figure 4.11: GPU Work Runtime Vs. Total Runtime on the *Sandy-Kepler* Node

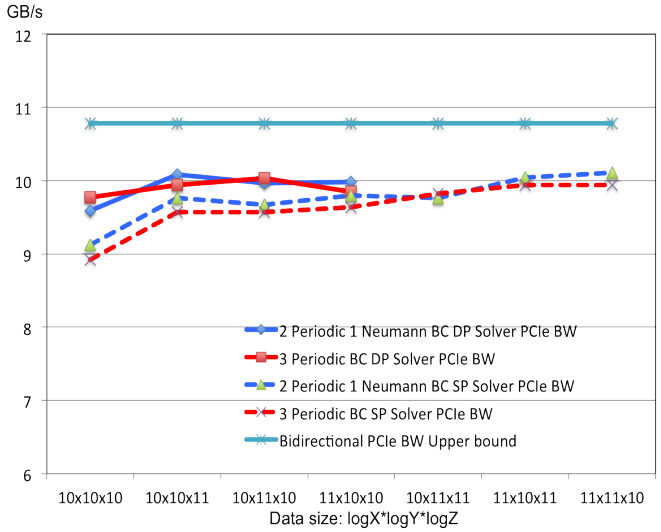


Figure 4.12: Bidirectional PCIe Bandwidth on the *Sandy-Kepler* Node

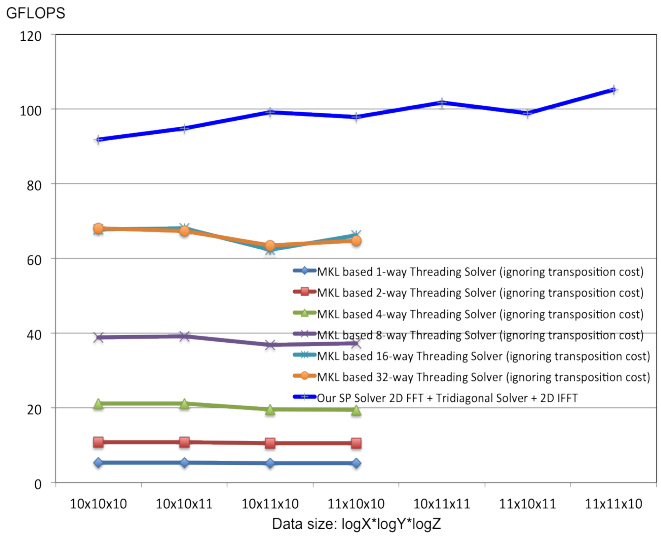


Figure 4.13: 2 Periodic 1 Neumann BC Single Precision Perf. on the *Sandy-Kepler* Node

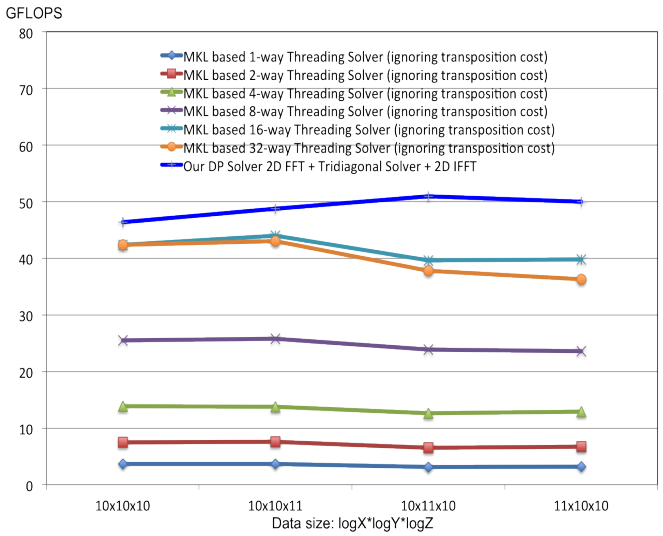


Figure 4.14: 2 Periodic 1 Neumann BC Double Precision Perf. on the *Sandy-Kepler* Node

Table 4.2: Compiler and Library configuration

Node	*Nehalem-Tesla*	*Sandy-Kepler*
CUDA driver	304.88	319.23
CUDA SDK	5.0	5.5
Intel Compiler & MKL Library	2011	2013

4.5 Performance

In this section, we present a summary of the performance tests that have been conducted on our CPU-GPU platforms.

In our tests, the problem size is a power of two in each of the three dimensions. We use input sizes that cannot be accommodated by the device memory alone.

Since the essence of our algorithms is based on either 3D or 2D FFT computations, we use the following well-known formula to estimate the FFT GFLOPS performance,

assuming that the execution of a one dimensional FFT on data size NX is t seconds:

$$GFLOPS = \frac{5 \cdot NX \cdot \log_2(NX) \cdot 10^{-9}}{t} \tag{4.1}$$

We compare the performance of our FFT implementations against implementations obtained by employing SIMD enabled OpenMP based 2D or 3D FFT routines using Intel's MKL library.

We also evaluate the effective PCIe bandwidth achieved using Formula 4.2 to get a sense about the performance of our CPU-GPU asynchronous streaming strategy.

$$BW = \frac{2 \cdot sizeof(element) \cdot NX \cdot NY \cdot NZ \cdot 2^{-30}}{t} \tag{4.2}$$

where $sizeof(element)$ is the number of bytes occupied by each data element - 8 bytes for a single precision complex number or 16 bytes for a double precision complex number. The factor of 2 captures the fact that we are moving the data from the CPU to the GPU and then back to the CPU. The time t used in the formula excludes the CPU runtime for the X dimensional forward and inverse FFT work - it starts from the moment that the CPU begins to copy data from the system host memory to the pinned memory for the asynchronous memory copy and ends at the moment that all the results are copied back into the system host memory. The performance numbers reported are the median performance of 5 runs for each data size and boundary condition combination.

4.5.1 The Case of the Three Periodic Boundary Conditions

Our three periodic BC Poisson solver consists of a forward 3D FFT, a scaling (division) step for each element of the intermediate 3D array, followed by an inverse 3D FFT. Therefore, the number of GFLOPS achieved by our algorithm can simply be calculated based on the 3D FFT GFLOPS formula. Since we do not include the intermediate division scaling step in our estimate, we under-estimate the performance of our algorithm. Specifically, if the total execution time on a 3D data set of size $NX \times NY \times NZ$ is t seconds, then its GFLOPS can be measured using the standard formula:

$$GFLOPS = \frac{2 \cdot 5 \cdot NX \cdot NY \cdot NZ \cdot [\log_2 (NX \cdot NY \cdot NZ)] \cdot 10^{-9}}{t} \tag{4.3}$$

The coefficient 2 in the above formula captures the forward and the inverse FFT.

Figures 4.9 and 4.10 illustrate the GFLOPS performance on the *Sandy-Kepler* node of our 3 periodic BC case Poisson solver and the combined 3D forward and inverse FFT using the MKL library for the single precision and double precision cases respectively. Figure 4.15 shows the GFLOPS performance on the *Nehalem-Tesla* node using single precision. Due to the Tesla C1060's relatively low performance of double precision floating point operations, we did not test our algorithms on the double precision version.

For the MKL library performance on each node, the performance improved with the number of threads up to the maximum number of physical cores available on the machine. We show only the curves corresponding to the best performance on our nodes. In particular, the performance numbers of using 8 and 16 threads are similar on the dual

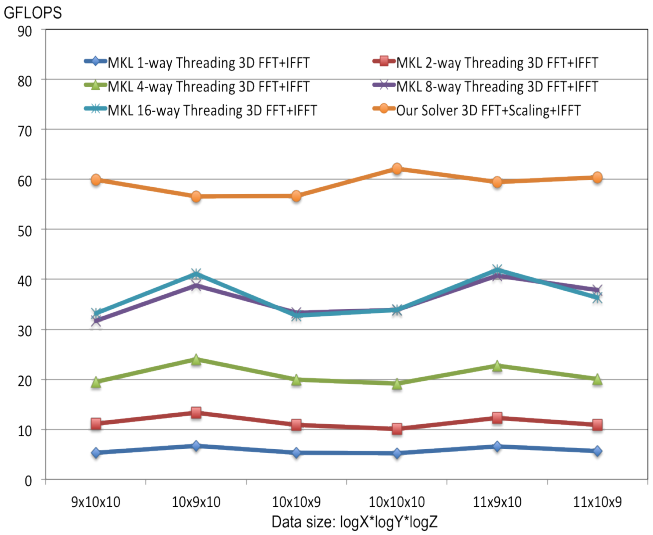


Figure 4.15: 3 Periodic BC Single Precision Perf. on the *Nehalem-Tesla* Node

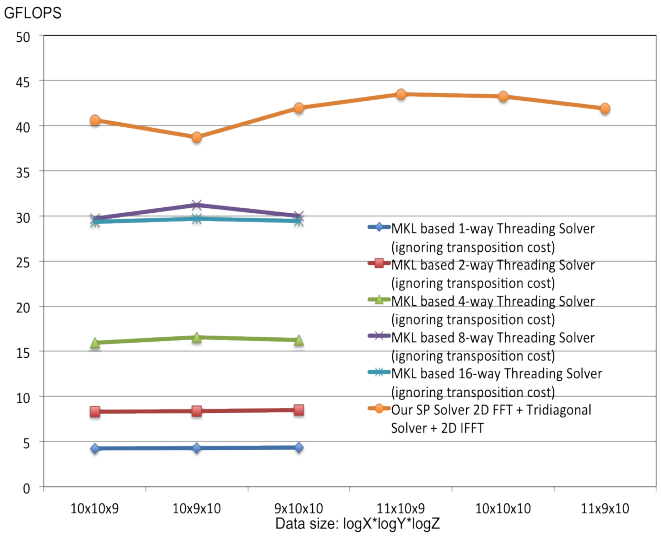


Figure 4.16: 2 Periodic 1 Neumann BC Single Precision Perf. on the *Nehalem-Tesla* Node

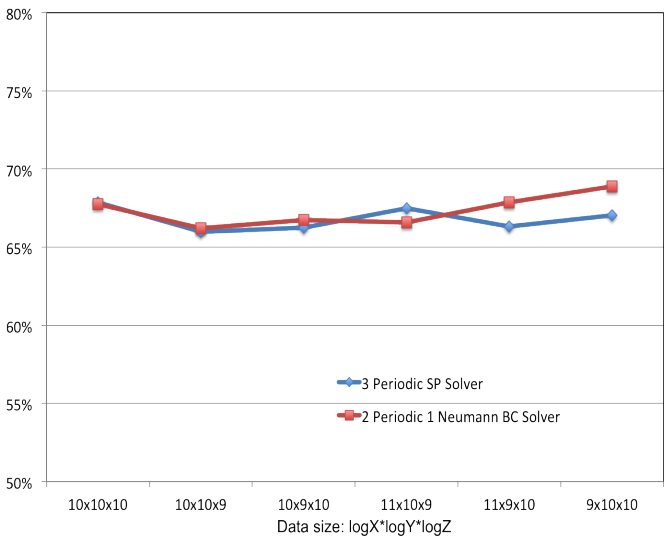


Figure 4.17: GPU Work Runtime Vs. Total Runtime on the *Nehalem-Tesla* Node

Figure 4.18: Effective PCIe Bandwidth on the *Nehalem-Tesla* Node

socket quad-core *Nehalem-Tesla* node's CPU and the performance numbers of using 16 and 32 threads are similar on the dual socket octa-core *Sandy-Kepler* node's CPU - both cases achieve the peak performance of the MKL library on our platforms.

A quick comparison shows that our Poisson solver, which includes 3D forward FFT, intermediate division scaling and 3D inverse FFT almost doubles the peak performance of the MKL library on the same node. A cross comparison of the single precision version and the double precision version on the *Sandy-Kepler* node shows that the single precision version is almost double the performance of the double precision version - which indicates the robustness of our CPU-GPU workload decomposition and that our implementation is indeed limited by the PCIe bus bandwidth.

In Figure 4.12 we illustrate the effective bidirectional PCIe bus bandwidth of the 3 periodic BC case on the *Sandy-Kepler* node: it ranges from 9GB/s to 10GB/s. We indicated the bidirectional bandwidth upper bound, which is the sum of pinned host memory to device memory bandwidth (5.44GB/s) and device memory to pinned host memory bandwidth (5.34GB/s) when the asynchronous memory copies are steady and completely overlapped. As mentioned before, when only single directional memory transfer is conducted, its performance is slightly better than concurrent memory transfer: the pinned host memory to device memory copy has a bandwidth of 5.7GB/s and the device memory to pinned host memory has a bandwidth of 6.2GB/s. Similarly, an average 4.5GB/s effective PCIe bus bandwidth is achieved on the *Nehalem-Telsa* node.

Figures 4.11 and 4.17 illustrate the effectiveness of the work decomposition on the *Sandy-Kepler* and the *Nehalem-Tesla* nodes respectively. As we can see from these figures, the runtime of the GPU related work - including the CPU memory copy work for

the GPU - constitutes more than 2/3 of the total runtime on both nodes. Recall that the execution rate of our GPU work is close to the PCIe bus bandwidth limit, and assuming more than one PCIe bus transfer is conducted, the additional runtime would surely exceed our CPU part work runtime.

4.5.2 The case of Two Periodic One Neumann Boundary Conditions

As described before, for the problem of size $NX \times NY \times NZ$, the two periodic and one Neumann BC case Poisson Solver consists of NZ number of 2D forward FFTs, each of size $NX \times NY$, $NX \times NY$ tridiagonal linear systems with matrix size $NZ \times NZ$, and NZ number of 2D inverse FFTs, each of size $NX \times NY$. We conduct a similar experimental tests as those carried out for 3 periodic BC case; however, we employ a GFLOPS formula that is appropriate for the corresponding computations. The number of GFLOPS now consist of two components: the 2D FFT computations, and the 1D tridiagonal solvers.

The 2D FFT or IFFT component can be easily captured as follows. If the execution time of 2D FFT or IFFT on data of size $NX \times NY$ is t seconds, then

$$GFLOPS = \frac{5 \cdot NX \cdot NY \cdot [\log_2 (NX \cdot NY)] \cdot 10^{-9}}{t} \tag{4.4}$$

The number of GFLOPS needed to solve a tridiagonal linear system of size N using Thomas algorithm is $8N$, and hence the total GFLOPS formula for the 2 periodic (say, the X and Y dimensions) 1 Neumann (say, Z dimension) BC is the following:

$$GFLOPS = \frac{NX \cdot NY \cdot NZ \cdot [10 \cdot \log_2 (NX \cdot NY) + 8] \cdot 10^{-9}}{t} \tag{4.5}$$

The total GFLOPS performance of our Poisson solver for this case is shown in Figure 4.13 (single precision-SP), Figure 4.14 (double precision-DP) and Figure 4.16 (SP). In this case, we are comparing the performance of our algorithm to the multi-threaded CPU version implementation based on OpenMP based MKL 2D DFT routines and a fairly optimized multi-threaded tridiagonal solver. The multi-threaded CPU implementation includes the following steps: 1) NZ batched execution of the 2D forward DFT of size $NX \times NY$; 2) transpose data from memory layout of $< x, y, z >$ to $< z, x, y >$; 3) solve $NX \times NY$ tridiagonal linear systems, each of NZ unknowns; 4) transpose memory layout from $< z, x, y >$ to $< x, y, z >$; 5)NZ batched execution of the 2D inverse DFT of size $NX \times NY$. The data transpositions of steps 2 and 4 are performed to enable better memory locality for the tridiagonal solver. Otherwise, the performance will be significantly worse. In order to capture an idealized lower bound of this optimized implementation, we did not include the runtime of the memory transposition when we calculate the GFLOPS performance. Note that in reality, no matter how the boundary conditions are aligned in x, y, z dimensions, poor memory locality would be experienced in one dimension or additional memory transpositions are necessary, which would degrade the performance of the CPU implementations significantly.

As we can see from the *Sandy-Kepler* node performance figures, our single precision complete solver is significantly faster than our idealized CPU version; however such advantage decreases as we convert to double precision. The reason for the advantage degradation is because for double precision, the same amount of data was transferred by the PCIe bus with half the FLOPS computation as that of the single precision version - with the vast compute power of the GPU under-utilized. However, considering the ex-

cluded transposition time, which could be quite significant, our solvers still show superior performance as a complete solver. Our solver naturally makes use of the memory locality of the X dimension FFT computation, and carefully eliminates the need of matrix transposition when utilizing the GPU in a vector-processor way. Similar conclusion can be drawn for the *Nehalem-Tesla* node - though it is single precision version, the advantage of using GPU is degraded by the restriction of single directional PCIe bus transfer and the relatively smaller best achievable bandwidth.

In terms of the PCIe bus bandwidth, Figures 4.12 and 4.18 indicate a good PCIe bandwidth for the 2 Periodic 1 Neumann BC case - for both single and double precisions on both nodes. Moreover, Figures 4.11 and 4.17 indicate our CPU-GPU work decomposition is quite general and effective for both the 3 periodic BC case and the 2 periodic 1 Neumann BC case.

4.6 Conclusion

We presented in this chapter a new strategy to map an FFT-based direct Poisson solver on a CPU-GPU heterogeneous platform, which optimizes the problem decomposition using both the CPU and the GPU. The new approach effectively pipelines the PCIe bus transfer and GPU work, almost entirely overlapping the CPU-GPU memory transfer time and the GPU computation time. Experimental results over a wide range of grid sizes have shown very high performance, both in terms of the number of floating point operations per second and the effective PCIe bus memory bandwidth. Our strategies were demonstrated equally effective across platforms and for different precision requirement.

Chapter 5: Out of Card Dense Matrix Multiplication Acceleration

5.1 Introduction

Clusters based on heterogeneous nodes currently are the trendy architecture for high performance computing. In this chapter, we present a dense matrix multiplication strategy specifically tailored for heterogeneous platforms in the case when the input is too large to fit on the device memory. Our strategy involves a CUDA stream based software pipeline that effectively exploits the hardware and software features of the CPU and the GPU thereby achieving, over a wide range of large data sizes, more than 95% of the best possible performance of the native CUDA DGEMM library. We adapt the blocking algorithms into a strategy that achieves near peak GPU computational rate within the bandwidth constraint of the PCIe bus bandwidth. By ensuring contiguous and near-peak-rate kernel execution flows, we were able to achieve more than 1 and 2 TFLOPS performance on a single node with dual socket multicore CPU using 1 and 2 GPUs respectively. Our results suggest the possibility of developing matrix computations on heterogeneous platforms which achieve native GPU performance on very large data sizes up to the capacity of the CPU memory.

Dense matrix operations are widely used as building blocks in many scientific and engineering problems. Double precision dense matrix multiplication (DGEMM),

constituting the most important routine of the LINPACK benchmark which is used to rank the top 500 supercomputers in the world, has been a major research focus in both academia and processor vendors. Currently clusters consisting of nodes based on multi-core CPU/many-core accelerators are very popular among the top 500 supercomputers list due to their peak FLOPS performance and their energy efficiency. Such architectures are likely to become more popular as we march toward the era of Exascale computing especially that power issues become quite critical. High performance native DGEMM libraries with 90% peak efficiency are often provided by vendors of CPUs [85], GPUs [86,87], and other accelerators such as Xeon Phi coprocessor [88]. However when it comes to the out of card performance, the great efficiency is typically compromised due to the substantial overhead caused by the memory transfers between the CPU and the accelerators.

In this chapter, we present a scalable scheme for accelerating DGEMM on heterogeneous CPU-GPU platforms, focusing on the case when the input is too large to fit on the device memory. Our scheme exploits hardware and software features of the CPU-GPU heterogeneous nodes and employ an asynchronous CUDA stream based software pipeline to achieve close to the best possible native CUDA BLAS DGEMM performance (CUDA BLAS assumes that both the input and output reside on the device memory).

The rest of the chapter is organized as follows. Section II provides an overview of the hardware and software features that are heavily used in this work, followed by a brief introduction of the most popular DGEMM libraries and related literature. Section III starts by discussing popular blocking schemes which are essential to high performance DGEMM followed by a description of our blocking scheme. Section IV provides details about our software pipeline which enables near peak performance. Section V illustrates

the performance of our strategy in terms of scalability.

5.2 Overview

Our target systems are CPU-GPU heterogeneous platforms consisting of multi-socket multi-core CPU and one or more GPU accelerators. The input data is much larger than the size of the device memory and is assumed to be initially held in the CPU memory. At the end of the computation, the output data must reside in the CPU memory as well.

We use two testbeds for our work. The first is a dual socket quad-core Intel Xeon X5560 CPU with 24GB main memory and two NVIDIA Tesla C1060 cards each with 4GB device memory - we refer to this testbed as the "*Nehalem-Tesla* node", after the codename of the CPU and the architecture of the GPU respectively. The second is a dual socket octal-core Intel Xeon E5-2690 with 128GB main memory and two NVIDIA Tesla K20 cards each with 5GB device memory - we refer to this testbed as the "*Sandy-Kepler* node" (we use Sandy rather than Sandy Bridge for brevity). The input data is much larger than the device memory and is assumed to be initially held in the CPU memory. At the end of the computation, the output data must reside in the CPU memory as well. Data transfers between the CPU main memory and the GPU device memory are carried out by PCIe Gen2x16 bus: unidirectional for the *Nehalem-Tesla* node (compute capability 1.3) and bidirectional for the *Sandy-Kepler* node (compute capability 3.5).

5.2.1 CUDA Programing Model

The CUDA programming model assumes a system consisting of a host CPU and massively data parallel GPUs acting as co-processors, each with its own separate memory [81]. The GPUs consist of a number of Streaming Multiprocessors(SMs), each of which containing a number of Streaming Processors (SPs or cores). The GPU executes data parallel functions called kernels using thousands of threads. The mapping of threads onto the GPU cores are abstracted from the programmers through - 1) a hierarchy of thread groups, 2) shared memories, and 3) barrier synchronization. Such abstraction provides fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism and this is based on similar hardware architecture among generations. Details of the CUDA programming model can be found at [81] and we will only refer to the aspects that are key to our optimization scheme. In this work, we are concerned with Tesla C1060 and K20 whose main features are summarized in Table 5.1. Note that, for the Tesla K20, the L1 cache and the shared memory per SM share a total amount of 64KB on-chip memory whose ratio is configurable as 1:3, 1:1 or 3:1.

5.2.2 PCIe bus

The CPU and the GPU communicate through the PCIe bus whose peak bandwidth is 8GB/s on PCIe Gen2x16 on both platforms. PCIe bus transfer typically uses pinned memory to get better bandwidth performance because the GPU cannot access data directly from the pageable host memory. A temporary pinned memory is implicitly used as a staging area. The bandwidth difference between using a pinned memory versus page-

Table 5.1: The Two GPUs Used in This Chapter

GPUs	Tesla C1060	Tesla K20
SMs per GPU	30	14
SPs per SM	8	192
Registers per SM	16K	64K
Shared Mem per SM	16KB	16KB-48KB
L1 Cache per SM	NA	48KB-16KB
L2 Cache per GPU	NA	1.25MB
Global Mem per GPU	4GB	5GB
GPU clock rate	1296MHz	706MHz
Memory clock rate	800MHz	2600MHz
Memory bandwidth	102.4GB/s	208GB/s
Compute Capability	1.3	3.5

able memory varies among platforms depending on whether both CPU and GPU support the same generation of the PCIe bus, their own DRAM bandwidth, etc.. For example, on our *Sandy-Kepler* node, the H2D bandwidth is around 3.3GB/s if we use pageable memory and similarly, the bandwidth of D2H is around 3GB/s; on the other hand, using pinned memory we can reach 5.7GB/s for H2D transfer and 6.3GB/s for D2H transfer. However, we should not over-allocate pinned memory so as not to reduce overall system performance but how much is too much is difficult to tell in advance and needs to be empirically determined. Another technique that is typically used is combining many small transfers into one large transfer to eliminate most of the per-transfer overhead. This is very practical for applications of large data size and when GPU device memory can only hold a subset of the dataset.

5.2.3 Asynchronous Streams

CUDA supports asynchronous concurrent execution between host and device through asynchronous function calls - control is returned to the host thread before the device has completed the requested task [81]. Data transfer and kernel execution from different CUDA streams can be overlapped when memory copies are performed between page-locked host memory and device memory. Some devices of compute capability of 2.x and higher (K20 in our evaluation) can perform memory copy from host memory to device memory (H2D) concurrently with a copy from device memory to host memory (D2H). With careful orchestration of the CPU work and CUDA streams, we essentially establish a CPU-GPU work pipeline of depth five in which computation and communication are organized in such a way that each GPU accelerator (K20) is always busy executing the kernel achieving 1TFLOPS performance. Since the data access pattern forces us to batch/pack small segments of data, we make use of the pinned memory to achieve better PCIe bus bandwidth rather than the pageable memory especially since as we need to use such a staging area anyway.

5.2.4 Existing CPU/GPU DGEMM Libraries

Almost all vendors have developed optimized DGEMM libraries that exploit the architectures of their processors quite effectively. The list includes the DGEMM libraries developed by Intel [89] and AMD for their multicore CPUs, the NVIDIA CUBLAS_DGEMM for the NVIDIA GPUs, and Intel's DGEMM library optimized for the Xeon Phi coprocessor. None of these libraries address heterogeneous platforms and each seems to have been

tailored for a particular architecture, even from generations to generations. [86] and [90] are such examples which optimized DGEMM for earlier CUDA Tesla architecture GPUs and later Fermi architecture GPUs, respectively.

5.3 Overall Matrix Multiplication Blocking Scheme

5.3.1 General Blocking Scheme

Blocking is a common strategy for most optimized DGEMM implementations which involves decomposing the matrices into blocks to fit into one or more levels of caches on a given CPU architecture.

5.3.2 Overview

The general double-precision matrix multiplication is defined as $C = \alpha AB + \beta C$, where A, B and C are $M \times K$, $K \times N$ and $M \times N$ matrices. Our DGEMM kernel assumes row-major format. The main strategy is to decompose the original CUBLAS DGEMM kernel into a set of outer-products as jobs to be assigned to asynchronous CUDA streams. The reasons for pursuing this approach are to: 1) accommodate the capacity difference between the GPU device memory and the CPU main memory; 2) alleviate the PCIe bus bandwidth requirement for a given computation requirement; 3) allow more $\{M, K, N\}$ combinations to benefit from the high FLOPS performance/memory ratio; and 4) assign "independent" jobs to CUDA streams to make use of the parallelism of the CPU, the PCIe bus and the GPU.

The DGEMM kernel can be decomposed as follows:

$$C_{ij} = \alpha \sum_{k=0}^{K/bk} A_{ik}^{(0)} B_{kj}^{(0)} + \beta C_{ij}^{(0)}, \tag{5.1}$$

where (0) indicates that the input data originally resides on the CPU, A_{ik}, B_{kj} and C_{ij} are sub-blocks of matrices A, B and C of sizes $bm \times bk$, $bk \times bn$ and $bm \times bn$ respectively.

Here, each job consists of computing a single C_{ij} sub-block. Assuming $s = K/bk$, the C_{ij} computation consists of s steps, with step s generating the final answer of the corresponding sub-block. We will refer to these steps as the s basic tasks for one job, that of computing C_{ij}. The staging of such computation can be represented in terms of the following formula.

$$\begin{aligned} C_{ij}^{(s)} &= \alpha^{(0)} \sum_{k=0}^{s} A_{ik}^{(0)} B_{kj}^{(0)} + \beta^{(0)} C_{ij}^{(0)} \\ C_{ij}^{(1)} &= \alpha^{(0)} A_{i0}^{(0)} B_{0j}^{(0)} + \beta^{(0)} C_{ij}^{(0)} \\ C_{ij}^{(k+1)} &= \alpha^{(0)} A_{ik}^{(0)} B_{kj}^{(0)} + C_{ij}^{(k)}, k = 1, ..., s \end{aligned} \tag{5.2}$$

That is, for each step k of each job C_{ij}, we compute the matrix multiplication of $C_{ij}^{(k+1)} = \alpha^{(0)} A_{ik}^{(0)} B_{kj}^{(0)} + \beta C_{ij}^{(k)}$, for $k = 0, ..., s-1$ with $\beta = 1$ for $k \neq 0$ steps and β equal to the original $\beta^{(0)}$ in the calling function when $k = 0$.

Such a decomposition offers for our platforms the following advantages:

1. The block sizes bm, bk and bn of A_{ik}, B_{kj} and C_{ij} allow us to apply fast native GPU DGEMM kernels, such as CUBLAS_DGEMM, which yields near peak *FLOPS* performance for a very broad range of problem sizes (that fit on the device memory).

, as we will ensure later, saturating the PCI-e bus bandwidth.

2. The result of step k is the input for step $k+1$ - this reduces the pressure on the PCIe bus as sub-matrix C_{ij} is reused. We only need to load $C_{ij}^{(0)}$ before the first step and store $C_{ij}^{(s)}$ after the last step. Hence, the cost of moving block C_{ij} is amortized and the PCIe bus bandwidth requirement is alleviated. Such block reuse of A_{ik}, B_{kj} and/or C_{ij} can be applied to the other $< M, N, K >$ scenarios.

3. Since separate streams are responsible for writing separate sub matrix C_{ij}, we avoid synchronization overhead among streams and make the decomposition strategy scalable - in fact, strongly scalable.

5.3.3 GPU Device Memory Blocking

In this section, we analyze the conditions that will "determine" the dimensions bm, bk and bn of the blocks A_{ik}, B_{kj} and C_{ij}. We say "determine" because it turns out there may be considerable flexibility in terms of the choices of bm, bk and bn which will achieve near-optimal performance throughput on the GPU for certain $< M, N, K >$ scenarios . Recall that our target platform is a CPU-GPU heterogeneous platform - with both the input and output data residing in the CPU main memory and using GPU(s) as accelerators. We make an analogy of this model to a 1-level caching system assuming a CPU based DGEMM implementation. We view the GPU(s) as the new "CPU": multi-socket CPU if we are using more than one GPU. Moreover, the GPU device memory can now be viewed as the 1-level cache with a speed equal to the PCIe bus bandwidth and a capacity equal to the GPU device memory.

As we noticed earlier, the matrix block C_{ij} is reused among the steps (aka, tasks) for the same job. However, we need to consider how the three matrix blocks are transferred through the PCIe bus into the GPU memory. Different from CPU caching, the GPU(s) has 1) a much larger "cache" size (5GB for Tesla K20 vs 256KB per core for Xeon E5-2690), 2) a much smaller memory bandwidth (5.7GB/s H2D and 6.3GB/s D2H v.s. 73GB/s for dual-socket Xeon E5-2690 STREAM benchmark), 3) a much higher experimentally peak DGEMM library *FLOPS* rate (1053 *GFLOPS* (CUBLAS 5.5) on Tesla K20 vs 320 *GFLOPS* on dual socket Xeon E5-2690). Therefore, the huge "caching GPU device memory" offers a number of possibilities to adapt the block dimensions bm, bk and bn so as to reach the near-peak GPU native DGEMM performance (1 *TFLOPS*) for various matrix dimensions of M, K and N.

The space needed to store the three matrix blocks is given by $8\,bytes \cdot (bm \cdot bn + bm \cdot bk + bn \cdot bk)$, and such data will be used to perform $2mkn$ floating point operations. To keep the GPU execution units at full speed, assuming a peak performance $GFLOPS_{peak}$ of CUBLAS_DGEMM (1.053 *TFLOPS*), the resulting PCIe bus host to device transfer bandwidth is:

$$BW_{req} = \frac{8 \cdot (bm \cdot bk + bm \cdot bn + bk \cdot bn) \times 2^{-30}}{\frac{2 \cdot bm \cdot bk \cdot bn \times 10^{-9}}{GFLOPS_{peak}}} \tag{5.3}$$

To develop an intuition into the PCIe bus bandwidth requirement stated above, let us first assume that $bm = bn = bk = dim$, which yields $BW_{req} = (12 \cdot 931.3/dim)\ GB/s$ which has to be $< 5.7GB/s$ (H2D). This inequality assumes that PCIe bus is not shared among GPUs which is the case in our testbed - each GPU is directly connected to a CPU via PCI-E as we are using dual-socket CPU. Otherwise, it may need to be adjusted according to

the target platform by simply dividing the number of GPUs that are sharing a single PCIe bus. Solving this inequality, we get $dim > 1960$ - which when rounded to $dim = 2000$, the required space for the three blocks is merely 0.09GB.

Next let's consider the more general case, that is, the blocking dimensions are distinct. This results

$$4 \times (\frac{1}{bm} + \frac{1}{bn} + \frac{1}{bk}) \frac{2^{-30}}{10^{-9}} \times \textit{GFLOPS}_{peak} < \textit{BW}_{peak} \tag{5.4}$$

Substituting our *Sandy-Kepler* platform's $\textit{GFLOPS}_{peak} = 1053$ and $\textit{BW}_{peak} = 5.7$, we get

$$\frac{1}{bm} + \frac{1}{bn} + \frac{1}{bk} < \frac{1}{688} \tag{5.5}$$

This provides an overall guideline for the block sizes - any block dimension smaller than 688 on that platform implies an under-utilization of the GPU kernels - how severe the under-utilization depends how bad the chosen block size is. Note that no matter what kind of blocking scheme and data reuse are employed, at least one block needs to be transferred from the host memory. This also indicates, if we would like to use CUDA GPUs to accelerate host-stored dense matrix multiplication, there is a minimum dimension requirement, for example, on Tesla K20, $min\{M, N, K\} > 688$ for achieving a good efficiency relative to the native CUBLAS/DGEMM.

In Figure 5.1 we evaluate the GFLOPS performance of the LAPACK_DGEMM on the CPU and the CUBLAS_DGEMM (CUDA 5.5) using one K20 on our platform. Giving the peak FLOPS performance of the CPU and GPU, the best DGEMM performances

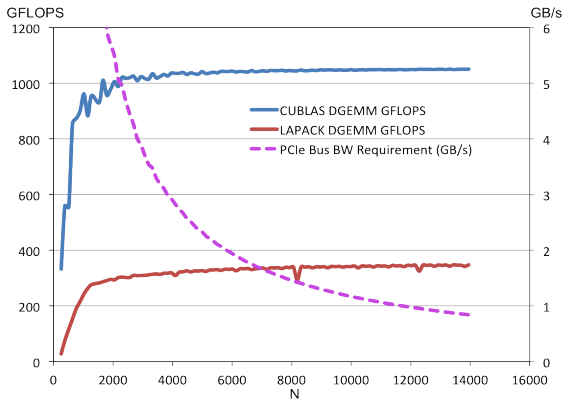


Figure 5.1: Library Benchmark and PCIe BW Requirement

(which is attainable for most reasonably large data sizes) on both CPU and GPU achieve more than 90% efficiency. Plugging the evaluated GFLOPS for a given square matrix size into the PCIe bandwidth requirement formula, we get the actual PCIe bus bandwidth requirement for that size, which is also plotted in Figure 5.1.

Since we are staging outer-product to compute the matrix block C_{ij}, the PCIe bus bandwidth requirement of later steps is alleviated compared to the first step as each time only matrix blocks A_{ik} and B_{kj} need to be loaded in later steps. Depending on the number of steps, if the chosen block dimensions bm, bn and bk only satisfy the PCIe bus requirement for two blocks, this would result in an idle period of GPU in the pipeline. From Figure 5.2, we can see that, as the square matrix size increases, the memory requirement on the PCIe bus drops rapidly. At the same time, we observe that as the matrix size increases, the required device memory space increases quadratically. This has several effects: 1) fewer number of concurrent jobs (computing individual C_{ij} blocks) can be scheduled as concurrent CUDA streams on the same GPU as different streams need separate space to store the matrix blocks; 2) the GPU idle time before its first stream starts to execute increases - since this is a computation-bound application, prolonging the GPU

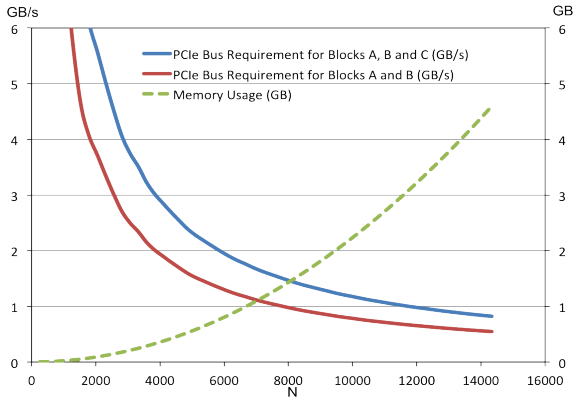


Figure 5.2: PCIe BW Requirement of Staging Outer-Product

idle time should be avoided if at all possible.

As a result, we list the rules to follow for selecting the block dimensions bm, bn and bk:

1. Using CUBLAS_DGEMM kernels to compute block matrix multiplication should achieve at least 1TFLOPS performance.

2. The space requirements for the matrix blocks A_{ik}, B_{ik} and C_{ij} should be large enough as stated in the formula above, but not be too large so that we are able to accommodate a number of concurrent CUDA streams to allow overlapping memory copy and kernel execution.

5.3.4 Packing and PCIe

Packing matrix blocks into micro- or macro- architecture friendly format is another popular technique used in optimized DGEMM. In addition to the size of each matrix block, the layout whether it is in row-major or column-major order is another issue to consider. In [85], Goto et al. packed matrix blocks that can be fit into the L2 cache to

achieve the minimal number of TLB entries. In [88], Heinecke et al. further extended the packing scheme to be the so-called "Knights Corner-friendly" matrix format (column-major format for sub-block A and row-major format for sub-block B) for their Xeon Phi coprocessor. The "Knights Corner" strategy achieves 89.4% efficiency.

Similarly, our strategy performs "packing" for matrix blocks during each step (task) of the outer-product of each job (C_{ij}). One of the main reason behind our "packing" scheme is to try to reach the peak experimental PCIe bus bandwidth. Recall that, on our *Sandy-Kepler* node, which is typical for a CPU-GPU heterogeneous platform, using pinned memory roughly doubles the bandwidth relative to using pageable memory. Such choices include smaller blocking dimensions and/or rectangular blocks which would incur larger PCIe bandwidth pressure given the same FLOPS count. Another key feature is that we take full control of the packing step, often using multi-threading memory copy. There are several reasons behind this choice. First, due to the memory capacity difference between the CPU main memory and the GPU device memory, we would need certain synchronization to avoid data hazards for the pinned memory - we make use of the CPU main thread to accomplish such synchronization. Second, using multi-threading could potentially improve the CPU system memory to pinned host memory copy bandwidth, as the CPU packing step could contribute to the overall runtime.

Such a packing step is mainly used to achieve high PCIe bus bandwidth - without packing, each step would require a good amount of small transfers. We make use of pinned host memory for such "multithreading" packing to ensure better PCIe bus bandwidth so as to offer better flexibility in terms of blocking size choices.

5.4 Multi-stage Multi-stream Software Pipeline

CUDA allows the use of streams for asynchronous memory copy and concurrent kernel executions to hide long PCIe bus latency [81]. A stream is a sequence of commands that execute in order; different streams may execute their commands out of order with respect to one another or concurrently. To obtain good performance, we need to overlap the execution of the kernels and the PCIe bus memory transfers from different streams to hide long device/host memory transfer latency. The PCIe bus memory transfer can be carried out between host and the device using pageable or pinned host memory. For the sake of performance, when the desired PCIe bus memory bandwidth is high, we explicitly allocate a reasonable amount of pinned host memory and use multi-threading to move data between the large pageable host memory and the the small pinned host memory.

We will start by discussing a typical five stage task that computes a matrix block multiplication. Then we describe how to organize multiple CUDA streams into a multi-stage multi-stream software pipeline. This will be followed by extending this software pipeline to accommodate different data reuse requirements, which will be based on the shapes of the matrices A, B and C so as to reduce the PCIe bus memory transfer amount.

5.4.1 A Simple Five-stage Task

We consider the task of computing $C^1_{ij} = A^0_{ik}B^0_{kj} + C^0_{ij}$, where matrix block sizes are $bm \times bn$, $bm \times bk$ and $bk \times bn$ respectively. One "*task*" here corresponds to one "*step*" for the job corresponding to the computation of C_{ij} as discussed in Section III.B. Such a task is completed by a single execution of a CUBLAS_DGEMM kernel call on the data

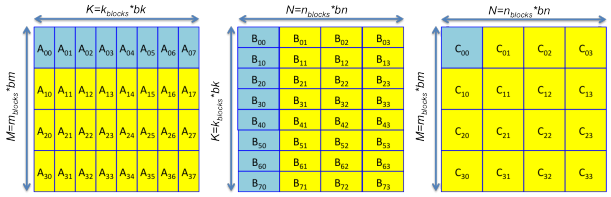


Figure 5.3: Matrix Blocking Scheme

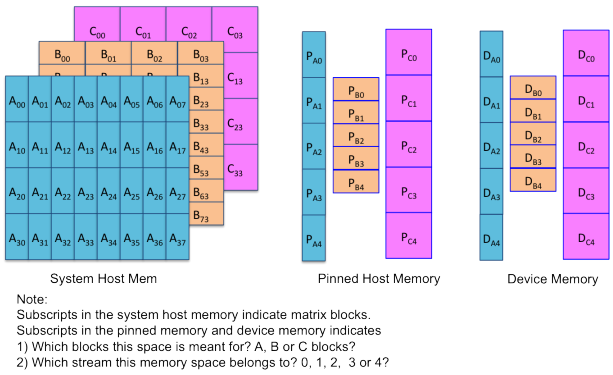


Figure 5.4: Memory Space Mapping (Assuming 5 Streams)

blocks that have been brought from the CPU host memory to the device memory via the pinned host memory. Once the kernel terminates, the result C_{ij}^0 is moved back to the CPU host memory via the pinned host memory. Specifically, this task can be divided into the following five stages:

1. Memory copy of blocks A_{ik}^0, B_{kj}^0 and C_{ij}^0 from the system host memory to the pinned host memory using multi-threading. We call this operation S2P memory copy.

2. Asynchronous CUDA memory copy from the pinned host memory to the device memory for blocks A_{ik}^0, B_{kj}^0 and C_{ij}^0. Such an operation is referred to as P2D memory copy.

3. CUBLAS_DGEMM kernel execution to compute $C_{ij}^1 = A_{ik}^0 B_{kj}^0 + C_{ij}^0$.

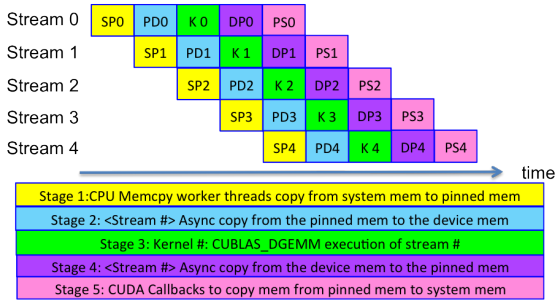


Figure 5.5: Basic 5-stage Pipeline

4. Asynchronous CUDA memory copy from the device memory to the pinned host memory for block C_{ij}^1. This operation will be referred to as D2P memory copy.

5. Memory copy of block C_{ij}^1 from the pinned host memory to the system host memory, possibly using multi-threading. This operation will be referred to as P2S memory copy.

To accomplish one five-stage task, we allocate pinned host memory and the device memory to hold blocks of C_{ij}, A_{ik} and B_{kj}. Assuming we can accommodate five independent tasks on the platform, the corresponding memory mapping is illustrated in Figure 5.4.

The time spent on each of the five stages can differ significantly depending on the block sizes, bus transfer bandwidth, and kernel performance, an issue that will be addressed next.

5.4.2 Basic Multi-stage pipeline

CUDA applications use asynchronous streams to hide the PCIe bus transfer time with kernel execution from different streams. For an *SN*-stage task with each stage consuming approximately the same amount of time t, we can allocate *SN* streams to handle

the tasks so as to get each task executed during a time period of length t time. As we assume a typical 5-stage tasks, we would expect to use in general 5 streams as illustrated in Figure 5.5.

We have to consider several factors that influence the duration of each stage of a task. First, we notice that stage 3 (Kernel execution stage) has to be the most time-consuming stage since our goal is match the native GPU DGEMM performance. We are of course assuming that this "time-consuming" stage is executed at optimal *FLOPS* throughput at the GPU near-peak performance, i.e., more than 1 *TFLOPS* for the K20 on DGEMM. This "time-consuming" characteristic is balanced by a good choice of block sizes bm, bn and bk, as determined by the inequality formula in the previous section.

Second, stages 1 and 5 are expected to be the faster than stages 2 and 4, due to the fact that the CPU DRAM bandwidth (in our case, 73GB/s [91]) is at least several times larger than the PCIe Gen2x16 bandwidth. That should be reasonable as long as there are no time gaps between the kernel executions among multiple streams, as we will show later.

5.4.3 Data Reuse in Multi-stage pipeline

Given a matrix multiplication problem, a straightforward approach would be assigning tasks-based jobs of C_{ij} to *SN* streams based on the decomposition formula in a given order.

$$C_{ij} = \alpha \sum_{k=0}^{K/bk} A_{ik}^{(0)} B_{kj}^{(0)} + \beta C_{ij}^{(0)} \tag{5.6}$$

Suppose we already have a reasonable blocking scheme as illustrated in Figure 5.3. bm, bn and bk for a problem size of M, N and K. We note $m_{blocks} = M/bm$, $n_{blocks} = N/bn$ and $k_{blocks} = K/bk$. We assign job_{id} as the computation of C_{ij} using the index mapping $job_{id} = i \times n_{blocks} + j$. Note that our index mapping attempts to minimize the TLB misses as the large input data need to be accessed from the CPU main memory. Let us assume that a number (*SN*) of CUDA streams with $stream_{id} = 0, ...,$ *(SN-1)* are executed concurrently. We assign the $m_{blocks} \times n_{blocks}$ jobs to the *SN* streams in a round-robin manner, modulo *SN*. For every assigned job, the stream is responsible for moving, computing and storing the final result of C_{ij} into the original host memory. The computation of C_{ij} involves a sequence of (k_{blocks}) of DGEMM function calls, that is, k_{blocks} basic tasks. We will describe later how what type of synchronization we need when we schedule consecutive jobs to the same stream. Figure 5.5 shows a simplified example of a 5-stage pipeline consisting of 5 CUDA asynchronous streams. In this example, each stream is handling a single job that includes a single task. Each stream uses its own pinned memory space and device memory space to store the A_{ik}, B_{kj} and C_{ij} blocks. ($k_{blocks} = 1$ in this figure.)

We use the matrix blocking scheme in Figure 5.3 as example to explain the resulting streams for the general multiple-task-per-job case ($k_{blocks} > 1$). According to our job_{id} and $stream_{id}$ relationship, we assign the computation of C_{00} to $stream_0$, which consists of 8 (k_{blocks}) iterations (sequence) of the basic five-stage stream tasks from $k = 0, ..., 7$.

As shown in Listing 5.1, the movement of block C_{ij} is guarded by conditional statements for data reuse. In general, at least one of the three blocks of A_{ik}, or B_{kj}, or C_{ij} may be reused.

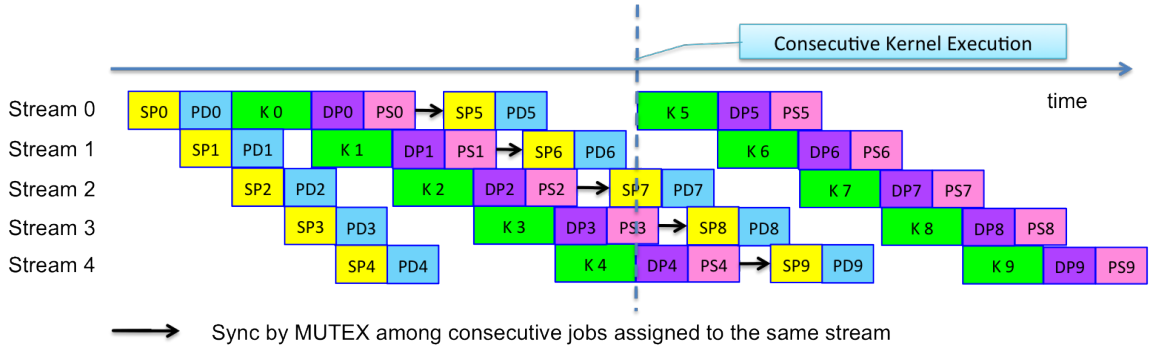


Figure 5.6: CPU-GPU Software Pipeline

```
1  for (int k = 0; k < 8; k++)
2  {
3     // stage 1
4     if(k==0) Memcpy_S2P_C(0,0);
5     Memcpy_S2P_A(0,k);
6     Memcpy_S2P_B(k,0);
7     // stage 2
8     if(k==0) Memcpy_P2D_C(0,0);
9     Memcpy_P2D_A(0,k);
10    Memcpy_P2D_B(k,0);
11    // stage 3
12    CUBLAS_DGEMM
13    // stage 4
14    if(k==7) Memcpy_D2P_C(0,0);
15    // stage 5
16    if(k==7)   Memcpy_P2S_C(0,0);
17 }
```

Listing 5.1: Tasks Per Job

5.4.4 Multi-stage Multi-stream Pipeline

The main goal in our design of the software pipeline is to ensure the continuous full utilization of the GPU near its peak performance. A key is to maintain a steady supply of data blocks to each GPU on our platform. Note that CUDA asynchronous streams can execute out of order with respect to each other but function calls within the

same stream have to execute in order. For matrix multiplication problem, we orchestrate the streams and their function calls in such a way that consecutive CUBLAS_DGEMM calls are executed immediately one after another, each resulting in near-peak performance per GPU. The overall scheduling of the multi-stage multi-stream pipeline is described in Listing 5.2. Note that we are able to achieve full utilization of the GPU for a much larger problem size than the device memory capacity using a memory mapping illustrated in Figure 5.4.

```
int jobs = m_blocks*n_blocks;
for (int i = 0; i < jobs; i +=SN)
{
  // tid = task_id;
  for(int tid=0; tid<k_blocks; tid++){
    // sid = stream_id;
    for(int sid=0; sid<SN; sid++) {
      job_id = i+sid;
      if(job_id>=jobs) break;
      // stage 1 (sync and schedule)
      Wait_for_CPU_S2P(job_id, tid);
      // stage 2
      Launch_AsyncMemcpy_P2D(job_id, tid);
      // stage 3
      CUBLAS_DGEMM(job_id, tid);
      // stage 4
      Launch_AsyncMemcpy_D2P(job_id, tid);
      // stage 5
      if(job_id+SN>=jobs)
      Update_last_flag(sid);
      Launch_CUDA_Callbacks_P2S(job_id, tid);
    }
  }
}
```

Listing 5.2: Multi-Stage Multi-Stream Pipeline

5.4.4.1 Synchronization

Memory reuse is used in our pipeline, which requires that mechanisms are put in place to avoid data hazards. We use MUTEX to achieve this goal. As we allocate different memory spaces for different streams, a data hazard can only happen within the same stream. We first assign flags for each stream in each potential block that can be overwritten (A, B and C respectively). These are each protected by a MUTEX after which we combine those flags appropriately to minimize the overhead of synchronization. Note that such synchronization overhead is typically "invisible' as long as it does not impede the CUBLAS_DGEMM executions as we have enough active CUDA streams to hide the synchronizations within the same stream's tasks/jobs as the black arrows illustrate in Figure 5.6. Specifically, we insert CUDA stream callbacks, executed as a CPU thread after previous CUDA kernel calls associated with that stream are completed. In the callbacks, we set the status flag to be "0" notifying the CPU worker threads to resume their memory copy work from the system host memory to the pinned host memory, which would flip the status to "1" and wait for the execution of another callback. A simplified pipeline with one task per job is illustrated in Figure 5.6.

5.4.5 Multi-stage Multi-stream Pipeline For Small K

Previously, we focused on matrix multiplication with relatively similar $< M, N, K >$ which gave us a significant number of choices for the block sizes. In this section, we tune the case, when M and N are much larger than K. That is, matrix A is skinny, matrix B is fat, and matrix C is large and almost square. This case is frequently used in parallel

dense matrix operations such as LU, QR and Cholesky factorizations. The challenge to the strategy described earlier is two-fold: 1) there is much less flexibility in selecting the block size for the dimension K; and 2) the large size of the input and output matrix C puts much more pressure on the bi-directional PCIe bus bandwidth than the square case.

As discussed before, in order to achieve near peak performance, no blocking dimension should be smaller than 688 for platforms using PCIe Gen2x16 bus. Hence, we assume $K > 688$ and we use $K = 1024$ as an example. Due to the inequality bound, we necessarily have $k_{blocks} = 1$, which yields this simple outer product $C_{ij} = A_{i0}B_{0j}$. This means for each CUBLAS_DGEMM kernel execution, we would have to load and store C_{ij} block, which is unavoidable. Even worse, due to the fact that K, aka bk is small, we are left with no choice but to have larger bm and bn to guarantee the inequality will still hold. As a result, this gives us a really big C_{ij} block to transfer bi-directionally in addition to the relatively smaller size A and B blocks.

We optimize such a situation in two ways. First, we assign jobs to the streams for which blocks of A or B could be reused in different jobs. For example, we assign the computation of C_{0j} to stream 0 and keep matrix A_{00} in the device memory throughout the computation of C_{0j} other than swap it out. Second, we use two components of the pinned host memory space for matrix C: one as Write-Combining Memory to conduct the H2D memory transfers for better bandwidth utilization; and the other one as default cacheable memory for the other way around as write-combining memory for D2H memory transfers.

Table 5.2: Compiler and Library configuration

Node	*Nehalem-Tesla*	*Sandy-Kepler*
CPU Name	Intel Xeon X5560	Intel Xeon E5-2690
Sockets x Cores	2x4	2x8
DRAM	24GB	128 GB
STREAM BW [91]	37GB/s	73 GB/s
icpc & MKL Lib	2013	2013
GPU Name	Tesla C1060	Tesla K20
Device Mem Size	4GB GDDR5	5GB GDDR5
Device Mem BW	102.4GB/s	208GB/s
SMs x SPs	30x8	13x192
PCIe bus	PCIe Gen2x16	PCIe Gen2x16
Bi-directional PCIe	No	Yes
PCIe achievable BW	5.4GB/s H2D 5.3GB/s D2H	5.7GB/s H2D 6.3GB/s D2H
CUDA driver	304.88	319.23
CUDA SDK	5.0	5.5
CUDA DGEMM Peak	75.3 GFLOPS	1053 GFLOPS

5.5 Performance

In this section, we evaluate the performance of our proposed multi-stage pipeline based approach on two different platforms. Detailed specifications of the platforms are listed in Table 5.2

5.5.1 Square Matrix Multiplication

The performance of our general blocking scheme for a range of matrix sizes from N=$1K$ to $52K$ on the *Sandy-Kepler* and *Nahalem-Tesla* nodes is shown in Figures 5.7 and 5.8 respectively. We compare the GFLOPS performance of our implementations using 1 and 2 GPUs as accelerators and Intel MKL multi-threading DGEMM using all the CPU cores on the *Sandy-Kepler* node.

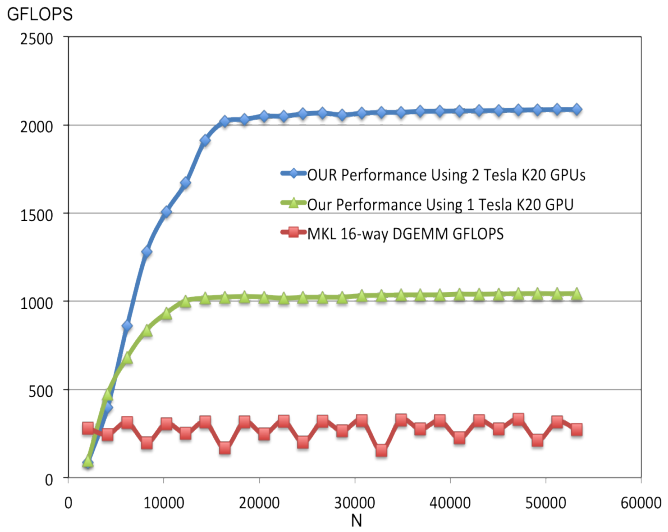


Figure 5.7: DGEMM Performance on *Sandy-Kepler* Node

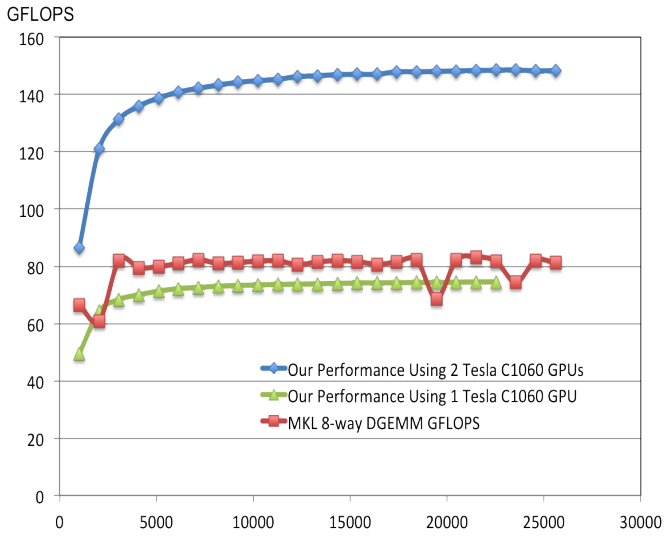


Figure 5.8: DGEMM Performance on *Nehalem-Tesla* Node

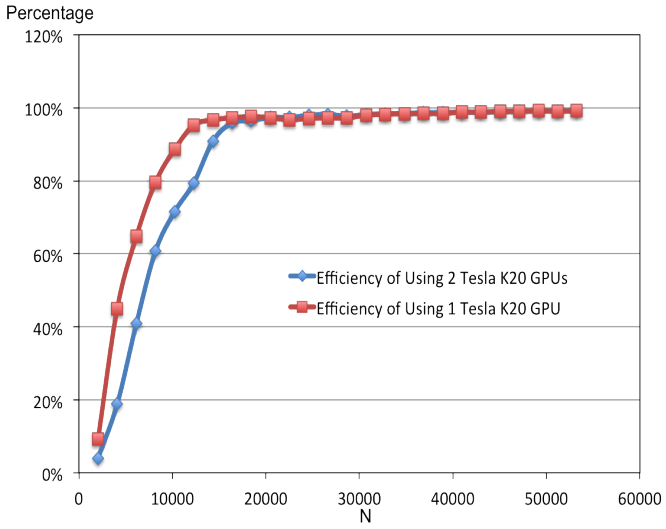


Figure 5.9: Efficiency on *Sandy-Kepler* Node

Similar to previous work, the number of FLOPS is determined by the expression $2 \cdot MNK$, where A is of size $M \times K$, B of size $K \times N$, and C of size $M \times N$. On the *Sandy-Kepler*, our approach greatly exploits the optimized performance of the CUDA DGEMM library and achieves 1 or 2 TFLOPS for all reasonably large data sizes by using either one or two GPUs. Such a performance is substantially better than the corresponding performance on the multi-core CPUs. In addition, unlike the native CUDA DGEMM library, whose problem size is limited by the device memory capacity, our approach essentially gives an illusion of a device memory size equal to the CPU host memory while delivering the same CUBLAS_DGEMM GFLOPS performance. In order to illustrate the generality of our scheme, we evaluate the same implementation on the *Nahalem-Tesla* node. Due to its weak double precision performance - a peak native library performance of 75.3GFLOPS - we are able to nearly match the native performance and double it for two GPUs.

To evaluate the effectiveness of our multi-stream software pipeline, we define the

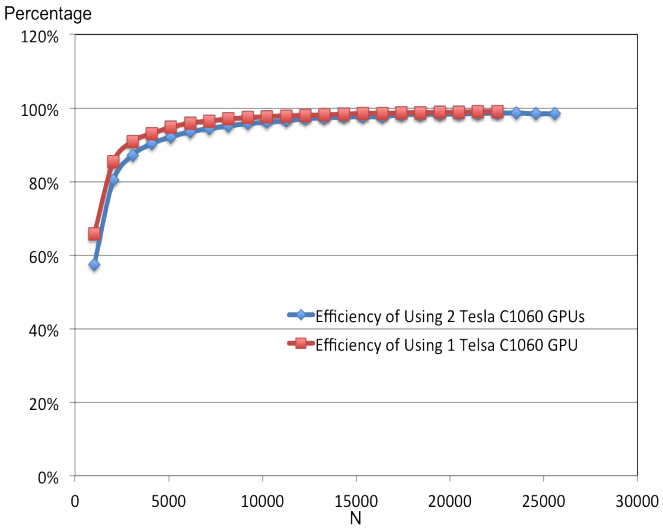


Figure 5.10: Efficiency on *Nehalem-Tesla* Node

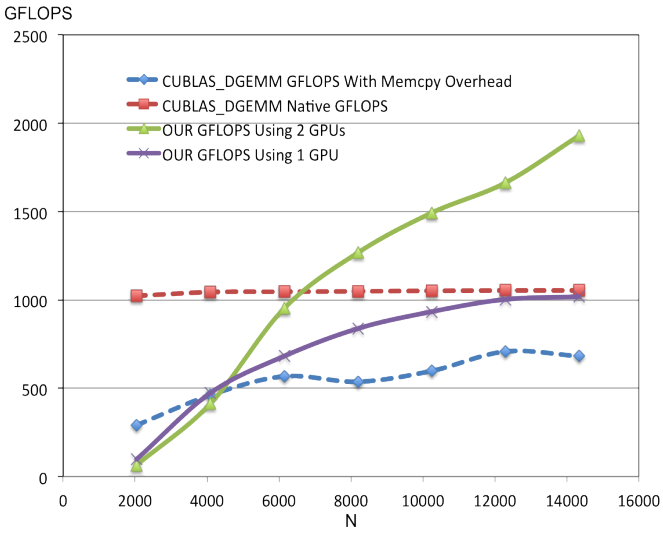


Figure 5.11: Smaller Size Performance on *Sandy-Kepler* Node

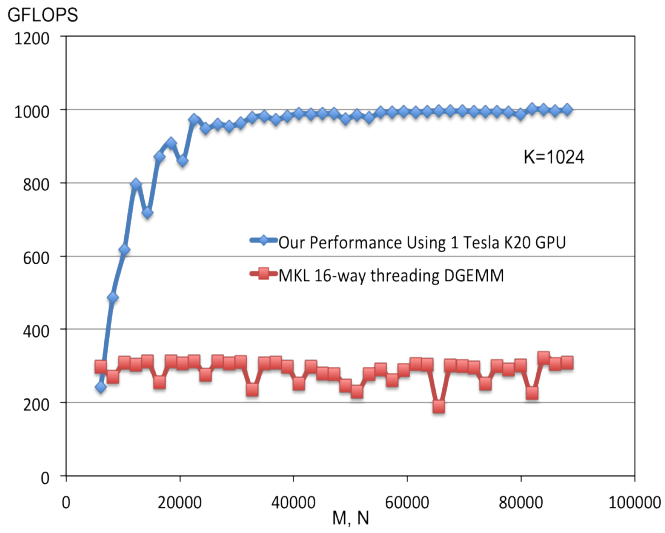


Figure 5.12: Small K DGEMM Performance *Sandy-Kepler* Node

efficiency as follows:

$$efficiency = \frac{GFLOPS_{achieved}}{GFLOPS_{peak\ lib\ perf}} \tag{5.7}$$

We demonstrate the efficiency of our scheme in Figures 5.9 and 5.10. As we can see from both figures, when the problem size is reasonably large, our software pipeline is quite efficient and brings almost all of the native CUDA DGEMM library performance out to the host memory. The same type of efficiency is obtained for both nodes in spite of their differences.

We also notice decomposition is not always beneficial for small data size, which was anticipated by our inequality bound. We demonstrate the performance of relatively smaller size matrices in Figure 5.11. Though the native CUBLAS_DGEMM performance on K20 is more than 1TFLOPS for all problem size of $N > 2K$, transferring the input from the CPU host memory and the output back to the CPU contribute a significant overhead. In fact, when the the problem size is fairly small, say $N = 2K$, we may simply

want to use a straightforward CUDA DGEMM call. Notice that in this case the problem fits on the device memory, while the focus of this section is on problems that cannot fit on the device memory.

5.5.2 Skinny A and Fat B Case

We now illustrate the performance for the case when matrix A is skinny and matrix B is fat. We fix $K = 1024$ and vary $M = N$ value over a wide range. Our strategy works extremely well and shows scalability similar to the square case.

Similarly, we demonstrate the GFLOPS performance and the efficiency as shown in Figure 5.12.

5.6 Conclusion

We have developed a pipelining strategy to carry out dense matrix multiplication for the case when the input size is much larger than the size of the device memory. Our strategy achieves almost the same native CUDA DGEMM library performance over a wide range of large sizes. We achieve more than 1 teraflops on a single GPU and twice the performance on two GPUs, thereby illustrating the possibility of using the GPUs with a memory size equal to the size of the main memory on the host machine. The key to this performance is the careful selection of the block sizes and the orchestration of the various stages of a CUDA multi-stream that ensures continuous GPU executions near peak performance. Our results raise the possibility of carrying out various dense matrix operations on very large matrices stored in the CPU memory while achieving native GPU

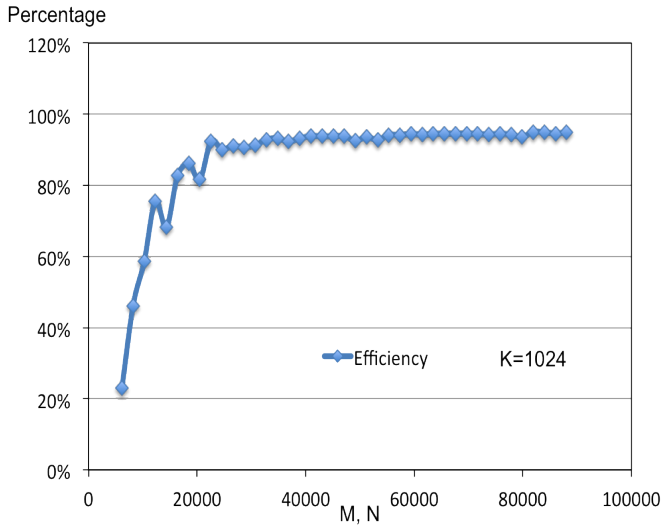


Figure 5.13: Small K DGEMM Efficiency on *Sandy-Kepler* Node

performance on matrices that fit on the device memory.

Chapter 6: Concluding Remarks and Future Perspectives

In this dissertation, we developed optimization techniques for mapping algorithms and applications onto CUDA GPU platform and CPU-GPU heterogeneous platforms for a number of demanding scientific applications.

We first addressed the problem of mapping multidimensional FFTs onto GPUs which resulted in extremely fast implementations for a wide number of data sizes across the Tesla and Fermi architectures. Our approach was carefully tailored to exploit the highly multithreaded environment in such a way as to almost completely overlap the FFT computations along the X dimension with the data transfers needed for the FFT computations along the other two dimensions. Moreover we minimized the number of global memory accesses while ensuring that each global memory access is a coalesced 128-byte memory transaction and optimizing the effects of related to partition camping, locality, and associativity. Our approach can easily be applied to 2D and 4D FFT computations to generate fast implementations on GPUs.

We also presented new approach to map an FFT-based direct Poisson solver on GPUs, which exploited the data parallel architecture of the GPUs Streaming Processors and the high device memory bandwidth that can be achieved through coalesced device memory transactions. The new approach used a novel strategy for computing three and

two-dimensional FFTs, while interleaving FFT computations along a dimension with other numerical computations required by the direct Poisson solver. Experimental results over a wide range of grid sizes have shown very high performance, both in terms of the number of floating point operations per second or the device memory bandwidth achieved by our algorithms. The performance numbers were superior to those that can be achieved using the CUDA FFT or the Nukada FFT Libraries in combination with well-known multi-threaded tridiagonal solvers.

We also ported the multi-dimensional FFTs and FFT-based direct Poisson solvers on CPU-GPU heterogeneous platforms. We minimized the data transfer on the PCIe bus connecting the CPU and the GPU as well as optimized the problem decomposition using both the CPU and the GPU. The new approach effectively pipelines the PCIe bus transfer and GPU work, almost entirely overlapping the CPU-GPU memory transfer time and the GPU computation time. Experimental results over a wide range of grid sizes have shown very high performance, both in terms of the number of floating point operations per second and the effective PCIe bus memory bandwidth. Our strategies were demonstrated equally effective across plat- forms and for different precision requirements.

Last, we extended our CPU-GPU heterogeneous software pipeline and presented a dense matrix multiplication strategy specifically tailored for heterogeneous platforms in the case when the input is too large to fit on the device memory. Our strategy achieves almost the same native CUDA DGEMM library performance over a wide range of large sizes. We achieved more than 1 teraflops on a single GPU and twice the performance on two GPUs, thereby illustrating the possibility of using the GPUs with a memory size equal to the size of the main memory on the host machine. The key to this performance

was the careful selection of the block sizes and the orchestration of the various stages of a CUDA multi- stream that ensures continuous GPU executions near peak performance. Our results raise the possibility of carrying out various dense matrix operations on very large matrices stored in the CPU memory while achieving native GPU performance on matrices that fit on the device memory.

A number of additional research questions are worth pursuing.

In Chapters 4 and 5, we developed a software pipeline for CPU-GPU heterogeneous platforms that demonstrated significant performance for two representative algorithms: memory-bound FFTs and computation-bound dense matrix multiplication. We expect a similar software pipeline could be extended to many other demanding applications such that the utilization of the heterogeneous node can be optimized. Many of the real-world problems involve sparsity and using CPU-GPU heterogeneous platforms to accelerate sparse applications is of fundamental importance [92].

Sparse matrix vector multiplication (SpMV) kernel is an important kernel used in many iterative methods in scientific, engineering and economic applications, as well as information retrieval. However, SpMV is often a bottleneck in that it demonstrates a low fraction of peak processor performance [93]. [94] has demonstrated that the SpMV kernel is a pure memory-bound problem, while GPUs are designed for computationally intensive work. A number of works [95] [96] have demonstrated success of SpMV implementations on GPU platforms or GPUs without considering the more practical heterogeneous platforms. With the PCIe bus bottleneck, the memory-bounded characteristics of SpMV are expected to be more severe for heterogeneous platforms and therefore it is certainly worth exploring the development of optimization strategies to solve this problem in het-

erogeneous platforms.

Graph-based algorithms are widely used in many domains such as scientific computing, social network analysis, data mining, etc. Optimizing the performance of such algorithms is non-trivial on heterogeneous platforms due to the fact that the memory access pattern is input-dependent, and can be quite irregular. Moreover, efficient parallel graph processing algorithms are expected to be even challenging [97] as their random memory access patterns do not benefit from the mainstream optimization techniques of parallel hardware architectures. There are efforts in developing special high-end systems for irregular applications such as graph processing featuring high memory bandwidth, very large memory capacity, and many cores that can be heavily multi-threaded [98]. Impressive performance of graph algorithms on such machines were reported [99] [100] but their popularity is very limited.

Breadth-first search (BFS) is a core primitive widely used in many graph algorithms and also serves as a core kernel in many graph benchmarks, such as Graph500 supercomputer benchmark [101]. Typically, a sparse graph is stored in the well-known Compressed Sparse Row (CSR) sparse matrix format and a significant amount of BFS work demonstrates great similarity from SpMV. Most of the GPU related work note such similarity and develop quadratic parallelizations, which are isomorphic to iterative SpMV in the algebraic semi-ring [102]. Notably, the work in [103] presents a CPU-GPU hybrid method to adapt the execution plans according to the graph features, which gives a marginal performance improvement for practical graph traversal but avoids some GPU-only worst cases. A notable drawback is that graph size of their work is limited by one GPU memory size and in their evaluation they ignore the data copy time from the CPU to the GPU. Unlike

others, [104] develops an asymptotically optimal BFS parallelization by fine-grained task management constructed from efficient prefix sums with 3.3 billions TE/s (1 GPU) and 8.3 billions TE/s (4 GPUs). However, despite the fact that their implementation scales up to four GPUs, linear scalability is not achieved. Moreover, it would be of very practical importance to actually utilize both the CPU and the GPU for work while hiding the PCIe bus data transfer since the moment we are adding additional data and work to the graph applications the problem size for GPU-only based implementations will be decreased immediately.

Bibliography

[1] John Shalf, Sudip Dosanjh, and John Morrison. Exascale Computing Technology Challenges. In *Proceedings of the 9th International Conference on High Performance Computing for Computational Science*, VECPAR'10, pages 1–25, Berlin, Heidelberg, 2011. Springer-Verlag.

[2] Gordon E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8), April 1965.

[3] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), 2005.

[4] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[5] Thomas Rauber and Gudula Rnger. *Parallel Programming - for Multicore and Cluster Systems.* Springer, 2010.

[6] Michael J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, September 1972.

[7] Arne Halaas, Børge Svingen, Magnar Nedland, Pål Sætrom, Ola Snøve, Jr., and Olaf René Birkeland. A Recursive MISD Architecture for Pattern Matching. *IEEE Trans. Very Large Scale Integr. Syst.*, 12(7):727–734, July 2004.

[8] Alfred Spector and David Gifford. The Space Shuttle Primary Computer System. *Commun. ACM*, 27(9):872–900, September 1984.

[9] Stuart Oberman, Greg Favor, and Fred Weber. AMD 3DNow! Technology: Architecture and Implementations. *IEEE Micro*, 19(2):37–48, March 1999.

[10] Intel. *Intel Hyper-Threading Technology: Technical User's Guide*. Intel Corporation, January 2003.

[11] James Laudon, Anoop Gupta, and Mark Horowitz. Interleaving: A multithreading technique targeting multiprocessors and workstations. *SIGOPS Oper. Syst. Rev.*, 28(5):308–318, November 1994.

[12] David Koufaty and Deborah T. Marr. Hyperthreading technology in the netburst microarchitecture. *IEEE Micro*, 23(2):56–65, March 2003.

[13] NVIDIA Corporation. NVIDIA CUDA C programming guide, 2014.

[14] Blaise Barney. Introduction to Parallel Computing. https://computing.llnl.gov/tutorials/parallel_comp/.

[15] Ken Kennedy, Charles Koelbel, and Hans Zima. The rise and fall of high performance fortran: An historical object lesson. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 7–1–7–22, New York, NY, USA, 2007. ACM.

[16] Gary W. Johnson. *LabVIEW Graphical Programming: Practical Applications in Instrumentation and Control*. McGraw-Hill School Education Group, 2nd edition, 1997.

[17] MATLAB. *Version 8.3 (R2014a)*. The MathWorks Inc., Natick, Massachusetts, 2014.

[18] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

[19] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[20] Chuck Pheatt. Intel® threading building blocks. *J. Comput. Sci. Coll.*, 23(4):298–298, April 2008.

[21] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, August 1995.

[22] Scott Oaks and Henry Wong. *Java Threads*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1997.

[23] Joseph JaJa. *An Introduction to Parallel Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[24] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2-3):72–109, 1994.

[25] Oak Ridge National Laboratory. PVM: Parallel Virtual Machine. http://www.csm.ornl.gov/pvm/.

[26] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.0, 09 2012. Chapter author for Collective Communication, Process Topologies, and One Sided Communications.

[27] Blaise Barney. Message Passing Interface. https://computing.llnl.gov/tutorials/mpi/.

[28] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[29] Microsoft. *GPGPU Computing Horizons: Developing and Deploying for Microsoft Windows*. 2010.

[30] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. OpenACC: First Experiences with Real-world Applications. In *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, pages 859–870, Berlin, Heidelberg, 2012. Springer-Verlag.

[31] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A Scripting-based Approach to GPU Run-time Code Generation. *Parallel Comput.*, 38(3):157–174, March 2012.

[32] Inc. The MathWorks. *Parallel Computing Toolbox: User's Guide*. The MathWorks, Inc., Natick, MA, USA, 2014.

[33] James Malcolm, Pavan Yalamanchili, Chris McClanahan, Vishwanath Venugopalakrishnan, Krunal Patel, and John Melonakos. ArrayFire: a GPU acceleration platform, 2012.

[34] Top500. Top 500 supercomputer sites. http://www.top500.org/, June 2014.

[35] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The linpack benchmark: Past, present, and future. concurrency and computation: Practice and experience. *Concurrency and Computation: Practice and Experience*, 15:2003, 2003.

[36] Steve Ashby, Pete Beckman, Jackie Chen, Phil Colella, Bill Collins, Dona Crawford, Jack Dongarra, Doug Kothe, Rusty Lusk, Paul Messina, and Others. The Opportunities and Challenges of Exascale Computing. *summary report of the advanced scientific computing advisory committee (ASCAC) subcommittee at the US Department of Energy Office of Science*, 2010.

[37] Green500. The green 500 list. http://www.green500.org/, Sep 2013.

[38] Jack Dongarra. Algorithmic and Software Challenges when Moving Towards Exascale, March 2013. Talk at International Supercomputing Conference in Mexico (ISUM 2013).

[39] Jim Demmel. Communication-Avoiding Algorithms for Linear Algebra and Beyond, May 2013. IPDPS'13 Keynote talk.

[40] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[41] Matteo Frigo, Steven, and G. Johnson. The design and implementation of FFTW3. In *Proceedings of the IEEE*, pages 216–231, 2005.

[42] Markus Pschel, Jos M. F. Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David Padua, Manuela Veloso, Robert W. Johnson, Markus Pschel, Jos M. F. Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David Padua, Manuela Veloso, and Robert W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing algorithms. *Journal of High Performance Computing and Applications*, 18:21–45, 2004.

[43] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using phipac: A portable, high-performance, ansi c coding methodology. In *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, pages 340–347, New York, NY, USA, 1997. ACM.

[44] Kathy Yelick. Autotuning in the Exascale Era, June 2011. Keynote talk.

[45] David E. Shaw, Martin M. Deneroff, Ron O. Dror, Jeffrey S. Kuskin, Richard H. Larson, John K. Salmon, Cliff Young, Brannon Batson, Kevin J. Bowers, Jack C. Chao, Michael P. Eastwood, Joseph Gagliardo, J. P. Grossman, C. Richard Ho, Douglas J. Ierardi, István Kolossváry, John L. Klepeis, Timothy Layman, Christine McLeavey, Mark A. Moraes, Rolf Mueller, Edward C. Priest, Yibing Shan, Jochen Spengler, Michael Theobald, Brian Towles, and Stanley C. Wang. Anton, a special-purpose machine for molecular dynamics simulation. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 1–12, New York, NY, USA, 2007. ACM.

[46] Volodymyr V. Kindratenko, Jeremy J. Enos, Guochun Shi, Michael T. Showerman, Galen W. Arnold, John E. Stone, James C. Phillips, and Wen-mei Hwu. GPU clusters for high-performance computing. pages 1–8, 2009.

[47] C. Baker, G. Davidson, T. M. Evans, S. Hamilton, J. Jarrell, and W. Joubert. High Performance Radiation Transport Simulations: Preparing for Titan. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 47:1–47:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[48] Yifeng Chen, Xiang Cui, and Hong Mei. Large-Scale FFT on GPU Clusters. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 315–324, New York, NY, USA, 2010. ACM.

[49] Akira Nukada, Kento Sato, and Satoshi Matsuoka. Scalable Multi-GPU 3-D FFT for TSUBAME 2.0 Supercomputer. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 44:1–44:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[50] Ping Tak Peter Tang, Jongsoo Park, Daehyun Kim, and Vladimir Petrov. A Framework for Low-communication 1-D FFT. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 42:1–42:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[51] Li-Wen Chang, John A. Stratton, Hee-Seok Kim, and Wen-Mei W. Hwu. A Scalable, Numerically Stable, High-Performance Tridiagonal Solver using GPUs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 27:1–27:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[52] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Des. Test*, 12(3):66–73, May 2010.

[53] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming. *Parallel Comput.*, 38(8):391–407, August 2012.

[54] OpenACC — directives for accelerators. http://www.openacc.org.

[55] James Cooley and John Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90):297–301, 1965.

[56] Alexandre Joel Chorin. A numerical method for solving incompressible viscous flow problems. *J. Comput. Phys.*, 135(2):118–125, August 1997.

[57] P. Moin and K. Mahesh. Direct Numerical Simulation: A Tool in Turbulence Research. *Annual Review of Fluid Mechanics*, 30(1):539–578, 1998.

[58] U Piomelli. Large-eddy simulation: achievements and challenges. In *Progress Aero. Sci. 35*, pages 335–362, 1999.

[59] R. W. Hockney. A fast direct solution of poisson's equation using fourier analysis. *J. ACM*, 12(1):95–113, January 1965.

[60] P.N. Swarztrauber. A Direct Method for the Discrete Solution of Separable Elliptic Equations. In *SIAM ournal on Numerical Analysis, 11*, pages 1136–1150, 1974.

[61] P.N. Swarztrauber. The Methods of Cyclic Reduction, Fourier Analysis and the FACR Algorithm for the Discrete Solution of Poisson's Equation on a Rectangle. In *SIAM Journal on Numerical Analysis, 19*, pages 490–501, 1977.

[62] Tuomo Rossi and Jari Toivanen. A Parallel Fast Direct Solver for Block Tridiagonal Systems with Separable Matrices of Arbitrary Dimension. *SIAM J. Sci. Comput.*, 20(5):1778–1793, April 1999.

[63] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Modern Software Tools for Scientific Computing. chapter Efficient management of parallelism in object-oriented numerical software libraries, pages 163–202. Birkhauser Boston Inc., Cambridge, MA, USA, 1997.

[64] Rajat Mittal and Gianluca Iaccarino. Immersed Boundary Methods. In *Ann. Rev. Fluid Mech. 37*, pages 239–261, 2005.

[65] Ruetsh, Greg and Micikevicius, Paulius. Optimizing Matrix Transpose in CUDA, 2011.

[66] George L. Yuan, Ali Bakhoda, and Tor M. Aamodt. Complexity Effective Memory Access Scheduling for Many-core Accelerator Architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 34–44, New York, NY, USA, 2009. ACM.

[67] V. Volkov. Better Performance at Lower Occupancy, 2010.

[68] Liang Gu, Xiaoming Li, and Jakob Siegel. An empirically tuned 2D and 3D FFT library on CUDA GPU. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 305–314, New York, NY, USA, 2010. ACM.

[69] Nukada. Nukada FFT Library website. http://matsu-www.is.titech.ac.jp/ñukada/nufft/, 2011.

[70] Jing Wu and Joseph JaJa. Optimized Strategies for Mapping Three-dimensional FFTs onto CUDA GPUs. In *Innovative Parallel Computing (INPAR)*. IEEE Press, 2012.

[71] NVIDIA Corporation. CUDA and Fermi Update, 2010.

[72] Jing Wu, Joseph JaJa, and Elias Balaras. An optimized fft-based direct poisson solver on cuda gpus. *IEEE Transactions on Parallel and Distributed Systems*, 25(3):550–559, 2014.

[73] Jing Wu and Joseph JaJa. High performance fft based poisson solver on a cpu-gpu heterogeneous platform. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 115–125, Washington, DC, USA, 2013. IEEE Computer Society.

[74] Jing Wu and Joseph JaJa. Optimized {FFT} computations on heterogeneous platforms with application to the poisson equation. *Journal of Parallel and Distributed Computing*, 74(8):2745 – 2756, 2014.

[75] Naga K. Govindaraju, Scott Larsen, Jim Gray, and Dinesh Manocha. A Memory Model for Scientific Algorithms on Graphics Processors. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[76] Akira Nukada, Yasuhiko Ogata, Toshio Endo, and Satoshi Matsuoka. Bandwidth Intensive 3-D FFT kernel for GPUs using CUDA. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 5:1–5:11, Piscataway, NJ, USA, 2008. IEEE Press.

[77] Akira Nukada and Satoshi Matsuoka. Auto-tuning 3-D FFT library for CUDA GPUs. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 30:1–30:10, New York, NY, USA, 2009. ACM.

[78] Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High Performance Discrete Fourier Transforms on Graphics Processors. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 2:1–2:12, Piscataway, NJ, USA, 2008. IEEE Press.

[79] Y. Dotsenko, S. Baghsorkhi, B. Lloyd, and N. Govindaraju. Auto-tuning of Fast Fourier Transform on Graphics Processors. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 257–266, New York, NY, USA, 2011. ACM.

[80] Liang Gu, Jakob Siegel, and Xiaoming Li. Using GPUs to Compute Large Out-of-Card FFTs. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 255–264, New York, NY, USA, 2011. ACM.

[81] NVIDIA Corporation. NVIDIA CUDA C Programming Guide, 2013.

[82] Paulius Micikevicius. Multi-gpu programming. *Nvidia Cuda webinars. http://developer. download. nvidia. com/CUDA/training*, 2011.

[83] Intel. Intel xeon processor e5-2600 product family, 2012. `https://www-ssl.intel.com/content/www/us/en/benchmarks/server/xeon-e5-hpc/xeon-e5-hpc-memory-bandwidth-stream.html?`

[84] Dominik Goddeke and Robert Strzodka. Cyclic Reduction Tridiagonal Solvers on GPUs Applied to Mixed-Precision Multigrid. *IEEE Trans. Parallel Distrib. Syst.*, 22(1):22–32, January 2011.

[85] Kazushige Goto and Robert A. van de Geijn. Anatomy of High-performance Matrix Multiplication. *ACM Trans. Math. Softw.*, 34(3):12:1–12:25, May 2008.

[86] Vasily Volkov and James W. Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 31:1–31:11, Piscataway, NJ, USA, 2008. IEEE Press.

[87] NVIDIA Corporation. *CUBLAS Library User Guide*. NVIDIA, v5.5 edition, 2013.

[88] Alexander Heinecke, Karthikeyan Vaidyanathan, Mikhail Smelyanskiy, Alexander Kobotov, Roman Dubtsov, Greg Henry, Aniruddha G. Shet, George Chrysos, and

Pradeep Dubey. Design and Implementation of the Linpack Benchmark for Single and Multi-node Systems Based on Intel Xeon Phi Coprocessor. *Parallel and Distributed Processing Symposium, International*, 0:126–137, 2013.

[89] Intel. Math Kernel Library. http://developer.intel.com/software/products/mkl/.

[90] Guangming Tan, Linchuan Li, Sean Triechle, Everett Phillips, Yungang Bao, and Ninghui Sun. Fast implementation of DGEMM on Fermi GPU. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 35:1–35:11, New York, NY, USA, 2011. ACM.

[91] John D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[92] I. S. Duff, Roger G. Grimes, and John G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Softw.*, 15(1):1–14, March 1989.

[93] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 38:1–38:12, New York, NY, USA, 2007. ACM.

[94] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[95] Bor-Yiing Su and Kurt Keutzer. clspmv: A cross-platform opencl spmv framework on gpus. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 353–364, New York, NY, USA, 2012. ACM.

[96] Xintian Yang, Srinivasan Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on gpus: Implications for graph mining. *Proc. VLDB Endow.*, 4(4):231–242, January 2011.

[97] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan W. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(1):5–20, 2007.

[98] Oreste Villa, Antonino Tumeo, Simone Secchi, and Joseph B. Manzano. Fast and accurate simulation of the cray xmt multithreaded supercomputer. *IEEE Trans. Parallel Distrib. Syst.*, 23(12):2266–2279, December 2012.

[99] David A. Bader and Kamesh Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2. In *Proceedings of the 2006 International Conference on Parallel Processing*, ICPP '06, pages 523–530, Washington, DC, USA, 2006. IEEE Computer Society.

[100] D. Chavarria-Miranda, A. Mrquez, Jarek Nieplocha, K. Maschhoff, and C. Scherrer. Early experience with out-of-core applications on the cray xmt. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, April 2008.

[101] Graph500. The graph 500 list. http://www.graph500.org/, June 2013.

[102] Michael Garland. Sparse matrix computations on manycore gpu's. In *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, pages 2–6, New York, NY, USA, 2008. ACM.

[103] Sungpack Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 78–88, Oct 2011.

[104] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. *SIGPLAN Not.*, 47(8):117–128, February 2012.