UNIVERSITY OF CALIFORNIA, SAN DIEGO

Domain-Specific Translator and Optimizer for Massive On-Chip Parallelism

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Didem Unat

Committee in charge:

Professor Scott B. Baden, Chair
Professor Xing Cai
Professor Andrew McCulloch
Professor Allan Snavely
Professor Daniel Tartakovsky
Professor Dean M. Tullsen

2012

The dissertation of Didem Unat is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____

_____

Chair

University of California, San Diego

2012

DEDICATION

To my mom, dad and sister.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

In the first place, I would like to thank my thesis advisor, Scott Baden, for his guidance, patience and support throughout this research. He is very generous with his time, allowed me to work at my own pace and encouraged my personal growth. His wisdom, principles, and commitment to the highest standards inspired me from the very beginning of this research. I am also grateful to my committee members: Xing Cai, Allan Snavely, Daniel Tartakovsky, Andrew McCulloch and Dean Tullsen. Xing Cai undertook to act as my mentor at Simula with a great interest and enthusiasm. His feedback has been always prompt and invaluable. Allan Snavely took time to provide much advice on my research direction. I am very thankful to Simula for the generous 4-year of funding for my research by the Center of Excellence grant from the Norwegian Research Council to the Center for Biomedical Computing at Simula Research laboratory. Although the financial support is hugely appreciated, the greatest benefit for me was the opportunity to visit the lab in Norway twice and work closely with researchers at the lab.

I would like to thank a number of people who were essential in my career path leading up to the Ph.D. program: Irfan Ahmad, Pinar Pekel, Can Ozturan, Reyyan Somuncuoglu, and Ilkay Boduroglu. I was lucky to share my office with Han, Yajaira, Tan, Pietro, and Alden and the visiting students; Mohammed, Alex, and Tor. They were very enthusiastic about having coffee and fro-yo breaks with me (especially when I got bored of writing my dissertation).

I would never have survived without the companionship of my friends. Special thanks goes to Tikir, who has been a “dost", brother, and a mentor. I would like to thank my coffee shop/hiking/biking/dining buddies; Amogh, Ahmet and Zibi. I am truly indebted to Marisol, Ozlem, Elif, and Mevlude for helping me through tough times over the years and to my roommates Sveta, Federico, and Erin. I gratefully acknowledge my neighbor’s orange cat, Phoenix, for being so fluffy and comforting. I owe sincere thankfulness to Ian for caring about me and keeping me company. Many thanks for all the memories.

Finally, I would like to thank my parents and my sister who are the most dear to me: Thank you for letting me travel halfway across the world to pursue my career.

Portions of this thesis are based on the papers which I have co-authored with others.

- Chapter 3, Chapter 4 and Chapter 6, in part, are a reprint of the material as it appears in International Conference on Supercomputing 2011 with the title “Mint: Realizing CUDA performance in 3D Stencil Methods with Annotated C" by Didem Unat, Xing Cai and Scott B. Baden. I was the primary investigator and author of this paper.

- Section 7.1 in Chapter 7 is based on the material as it partly appears in Computing Science and Engineering Journal 2012 with the title “Accelerating an Earthquake Simulation with a C- to-CUDA Translator" by Jun Zhou, Yifeng Cui, Xing Cai and Scott B. Baden. Section 7.2 in Chapter 7 is based on the material as it partly appears in Proceedings of the 4th Workshop on Emerging Applications and Many-core Architecture 2011, with the title “Auto-optimization of a Feature Selection Algorithm" by Han Suk Kim, Jurgen Schulze and Scott B. Baden. Section 7.3 in Chapter 7 is based on the material as it partly appears in State of the Art in Scientific and Parallel Computing Workshop 2010 with the title “Optimizing the Aliev-Panfilov Model of Cardiac Excitation on Heterogeneous Systems" by Xing Cai and Scott B. Baden. I was the primary investigator and author of these three papers.

- Chapter 5, in part, is currently being prepared for submission for publication with Xing Cai and Scott B. Baden. I am the primary investigator and author of this material.

VITA

2006	Bachelor of Science, Boğaziçi University, İstanbul
2009	Master of Science, University of California, San Diego
2012	Doctor of Philosophy, University of California, San Diego

PUBLICATIONS

D. Unat, J. Zhou, Y. Cui, X. Cai, and S. B. Baden. “Accelerating an Earthquake Simulation with a C- to-CUDA Translator", *Computing in Science and Engineering Journal, Special Issue on Scientific Computing with GPUs*, 2012.

H. S. Kim, D. Unat, S. B. Baden, and J. P. Schulze. “Interactive Data-centric Viewpoint Selection", *Conference on Visualization and Data Analysis*, Burlingame, CA, 2012.

D.Unat, X.Cai, and S. Baden. “Mint: Realizing CUDA performance in 3D Stencil Methods with Annotated C", *International Conference on Supercomputing*, Tucson, AZ, 2011.

D. Unat, H. S. Kim, J. P. Schulze, and S. B. Baden. “Auto-optimization of a Feature Selection Algorithm", *Proceedings of the 4th Workshop on Emerging Applications and Many-core Architecture*, San Jose, CA 2011.

D. Unat, X. Cai, and S. Baden. “Optimizing the Aliev-Panfilov Model of Cardiac Excitation on Heterogeneous Systems", *Para 2010: State of the Art in Scientific and Parallel Computing*, Reykjavik, Iceland, 2010.

D. Unat, T. Hromadka III, and S. Baden. “An Adaptive Sub-Sampling Method for in-memory Compression of Scientific Data", *Data Compression Conference*, Snowbird, Utah, 2009.

ABSTRACT OF THE DISSERTATION

Domain-Specific Translator and Optimizer for Massive On-Chip Parallelism

by

Didem Unat

Doctor of Philosophy in Computer Science

University of California, San Diego, 2012

Professor Scott B. Baden, Chair

Future supercomputers will rely on massive on-chip parallelism that requires dramatic changes be made to node architecture. Node architecture will become more heterogeneous and hierarchical, with software-managed on-chip memory becoming more prevalent. To meet the performance expectations, application software will undergo extensive redesign. In response, support from programming models is crucial to help scientists adopt new technologies without requiring significant programming effort.

In this dissertation, we address the programming issues of a massively parallel single chip processor with a software-managed memory. We propose the Mint programming model and domain-specific compiler as a means of simplifying application development. Mint abstracts away the programmer's view of the hardware by providing a high-level interface to low-level architecture-specific optimizations. The Mint model requires modest recoding of the application

and is based on a small number of compiler directives, which are sufficient to take advantage of massive parallelism.

We have implemented the Mint model on a concrete instance of a massively parallel single chip processor: the Nvidia GPU (Graphics Processing Unit). The Mint source-to-source translator accepts C source with Mint annotations and generates CUDA C. The translator includes a domain-specific optimizer targeting stencil methods. Stencil methods arise in image processing applications and in a wide range of partial differential equation solvers. The Mint optimizer performs data locality optimizations, and uses on-chip memory to reduce memory accesses, particularly useful for stencil methods.

We have demonstrated the effectiveness of Mint on a set of widely used stencil kernels and three real-world applications. The applications include an earthquake-induced seismic wave propagation code, an interest point detection algorithm for volume datasets and a model for signal propagation in cardiac tissue. In cases where hand-coded implementations are available, we have verified that Mint delivered competitive performance. Mint realizes around 80% of the performance of the hand-optimized CUDA implementations of the kernels and applications on the Tesla C1060 and C2050 GPUs. By facilitating the management of parallelism and the memory hierarchy on the chip at a high-level, Mint enables computational scientists to accelerate their software development time. Furthermore, by performing domain-specific optimizations, Mint delivers high performance for stencil methods.

Chapter 1

Introduction

Within the next decade, it is hypothesized that High Performance Computing systems will contain exascale computers (10^{18} operations/second) operating with a power budget of no more than 20 MW [BBC+08]. This power constraint will dictate drastic changes in architectural design. In order to achieve the exascale power and performance goals, it is projected that the biggest architectural change will occur within the node, rather than across nodes [SDM11]. In particular, a node architecture will become more heterogeneous with specialized computing units and will contain unconventional memory subsystems with incoherent caches and software-managed memory. We have already started seeing concrete instances of such node design in current supercomputers that combine a general-purpose processor with a special-purpose massively parallel processor such as Graphics Processing Units (GPUs).

The performance benefits of the heterogeneous architectural design come at the expense of software development time because they confound the application programmer with tradeoffs. In order to take advantage of the architecture, the programmer has to understand the subtleties of heterogeneity in compute resources and the partially exposed memory hierarchy , resulting in significant implications for programming. Unlike conventional cache architectures, a software-managed memory requires the programmer to adopt unfamiliar programming style to explicitly orchestrate the data decomposition, allocation and movement in the software-level. Given the anticipated need for remapping applications to exascale systems, support from programming models is crucial.

This thesis addresses the programming issues of a massively parallel single chip processor with a software-managed memory and proposes a programming model, called Mint. The model abstracts the machine view from the programmer, and facilitates the mapping from the

high-level interface to low-level architecture-specific optimizations. The model is based on compiler directives, requiring modest recoding. Mint employs just five directives to annotate input programs, and we demonstrated that these are sufficient to accelerate applications.

We have implemented the Mint programming model for a system using the Nvidia GPUs as the massively parallel single chip processor. The Mint source-to-source translator for GPUs accepts traditional C source with Mint annotations and generates optimized CUDA C. The translator parallelizes loop-nests, manages data movement and threads. It includes a domain-specific optimizer that targets stencil methods, an important problem domain with a wide range of applications such as partial differential equation solvers. By restricting the compiler optimizations to a certain problem domain, we can more easily allow guided and specialized performance optimizations. The Mint optimizer performs data locality optimizations and uses on-chip memory (e.g registers and shared memory) to reduce memory accesses, particularly useful for stencil methods. Thus, the optimizer delivers higher performance for stencil methods compared to general purpose compilers [LME09, Wol10a, UCB11].

The challenge when designing a custom programming interface for an application domain is to come up with a small set of parameters that significantly affect performance. We designed Mint in a way that these parameters are administered by *clauses* and *compiler options*, which are controlled by the programmer at a high level. The principal effort for the Mint programmer is to identify time-consuming loop nests and annotate them with tunable clauses. These clauses govern data and workload decomposition across threads and enable non-experts to tune the code without entailing disruptive reprogramming. In addition, the compiler options give the programmer the flexibility to explore different optimizations applied to the same program, saving significant programming effort. Even though our compiler targets the Nvidia GPUs, the programming model can be implemented for other massively parallel architectures that have heterogeneous compute resources and software-managed memory hierarchy. The optimization strategies of stencil methods are essentially the same but the implementation of the clauses and compiler options exhibits differences based on the target architecture.

In this thesis, we demonstrate the effectiveness of the Mint model and its compiler for a set of widely used stencil kernels. The representative kernels share a similar communication and computation pattern with large applications, but are less complex to analyze. Mint realized 80% of the performance obtained from aggressively hand-optimized CUDA on the 200- and 400-series of Nvidia GPUs. We believe this performance gap is reasonable in light of Mint's reduced learning curve compared with extensive changes needed to port the code by hand.

Mint is not only useful for simple stencil kernels, but also in enabling acceleration of whole applications. The first application that we accelerated by using Mint is the AWP-ODC, an earthquake-induced seismic wave propagation code. The resultant Mint-generated code realized about 80% of the performance of the hand-coded CUDA. The second application is a computer visualization algorithm which detects features in volume datasets. Mint enabled the algorithm to realize real time performance in 3D images, which previously had been intractable on conventional hardware. Lastly, we studied the Aliev-Panfilov system which models signal propagation in cardiac tissue. For the cardiac simulation, Mint achieved 70.4% and 83.2% of the performance of the hand-coded CUDA in single and double precision arithmetic, respectively.

Thesis Contributions

- We have introduced the Mint programming model, an annotation-based interface to facilitate programing on the massively parallel multicore. The model employs only five directives and requires modest amount of programming effort to accelerate applications. Mint programming model facilitates programming for computational scientists, so that they can smoothly adopt technologies that rely on massive parallelism on a chip.

- We have implemented a source-to-source translator and optimizer which implements the Mint model for the GPU-based systems. The translator relieves the programmer of a variety of tedious tasks such as managing device memory and constructing device kernels. The domain-specific optimizer detects the stencil structure in the computation and performs data locality optimizations using the on-chip memory resources.

- We have demonstrated the effectiveness of the Mint model and the compiler on commonly used stencil kernels as well as real-life applications. Mint generated codes realize about 80% of the performance of hand-coded CUDA.

- Finally, we have discussed the applicability of Mint to future architectures. Future architectures may or may not have GPUs as the building blocks but will have a high degree of on-chip parallelism and software-managed memory hierarchy. Mint can be tailored to address programmability issues of future systems.

Thesis Outline

- Chapter 2 provides motivation and background for the thesis. It describes the trend in computer architecture and processor technologies, and also discusses the characteristics

of scientific applications in general and stencil-based applications in depth. The chapter presents available programming models, languages and interfaces for the current systems.

- Chapter 3 introduces the Mint programming model. Before introducing the interface of the model, we discuss the underlying hardware assumptions. The chapter continues with the Mint execution and memory model. Next, it presents the details of each directive and provides a simple example to illustrate the purpose of the directives.

- Chapter 4 discusses the source-to-source translator that implements the Mint model. The translator has two main stages. This chapter presents the first stage, *Baseline Translator*, which transforms C source code with Mint annotations to unoptimized CUDA, generating both device and host codes.

- Chapter 5 discusses the second stage of the translator, which is the the domain-specific optimizer targeting stencil methods. This chapter provides the details about the stencil analysis and on-chip memory optimizations to improve data locality. It gives several generated-code examples to demonstrate the impact of the compiler optimizations. To let the reader better comprehend the compiler optimizations, the chapter begins with an overview of the general optimization strategies for stencil methods.

- Chapter 6 demonstrates the effectiveness of the Mint translator by studying a set of widely used stencil kernels in two and three dimensions. The chapter first provides a discussion on the computer testbeds and software used throughout the thesis, then presents the performance results for commonly used stencil kernels.

- Chapter 7 presents case studies that validate the effectiveness of Mint on real-world applications coming from different problem domains. The first application is a cutting-edge seismic modeling application. The second comes from computer vision, the Harris interest point detection algorithm. The third study is a 2D Aliev-Panfilov model that simulates the propagation of electrical signals in the cardiac cells. After presenting background for each application, the chapter discusses the Mint implementation and the performance of the generated code along with performance tuning efforts.

- Chapter 8 discusses the limitations of the Mint model, future directions and concludes the dissertation.

Chapter 2

Motivation and Background

This chapter discusses why massive parallelism on a chip is inevitable and why software tools are needed to program them. It provides the background for the architectural trends and emerging massively parallel chip technologies in High Performance Computing in Section 2.1. We give special attention to Graphics Processing Units (GPUs), which provide an abundance of on-chip parallelism and have the potential to become the building blocks of an exascale system. Section 2.2 describes the common patterns in scientific applications with an emphasis on stencil computation. Section 2.3 presents programming environments in High Performance Computing systems such as libraries, languages, and programming models in the context of multicore processors.

2.1 Trends in Computer Architecture

Gordon Moore stated that the number of transistors per integrated circuit will double every 18-24 months [Moo00]. Even though Moore initially predicted that the trend would continue for at least 10 years, his prediction guided semiconductor technology for almost half a century until chip vendors hit the “power wall" [Mud01]. The power consumption of processors exponentially increased, as processor designers kept increasing the microprocessor clock speeds and added more instruction-level parallelism (ILP) through pipelining, out-of-order execution and superscalar issue [OHL07, KAB$^+$03]. Consequently, at the beginning of the 21st century, chip designers have started looking for new ways to improve processor performance. They switched to a new design paradigm, which integrates two or more processing cores on a single computing component. The chip area that used to be covered by a single large processor is now filled

by a collection of smaller processors. The new technology, called *multicore*[1], quickly became prevalent. According to the Top500 supercomputer rankings [Top], today more than 80% of the supercomputers rely on multicore processors.

The composition, purpose, and number of cores in multicore architectures show great variety. It is difficult to create a taxonomy because family members have distinct design features. We divide the multicore architectures into two categories based on their design philosophies even though the two categories may converge over time. The first is the general purpose multicore processor that is capable of running a wide variety of applications including the operating system. We will refer to this group as *general-purpose multicore* processors. Members of this group employ relatively complex CPUs, focus on single thread performance, and utilize ILP to some extent, compared to the second category, *massively parallel single chip* processors. A massively parallel chip consists of a relatively large number of simple and low power cores, tailored towards throughput computing with less focus on single-thread performance. In the next section we briefly go over the general-purpose multicore architectures and then introduce massively parallel single chip processors.

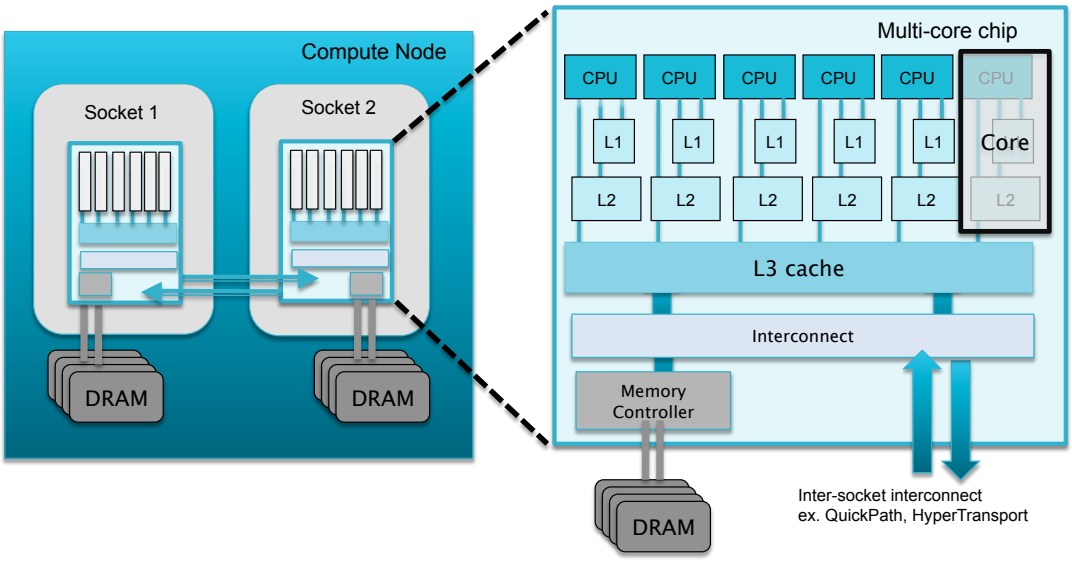


Figure 2.1: Abstract machine model of a compute node containing two general-purpose multicore chips on two sockets, each with 6 cores.

[1] In this thesis, we use the term “multicore", to refer to cores manufactured on the same integrated circuit die.

2.1.1 General-Purpose Multicore Processors

General-purpose multicores provide fast response time to a single thread. They are capable of running a wide variety of applications including operating systems and databases. Today, most high-end multicores are limited up to 10 cores (about 12 cores on a 2 die multi-chip module). Fig 2.1 shows an abstract machine model of a node containing two multicore chips, each with 6 cores. In a typical multicore chip, cores have private L1 and L2 caches but share L3 cache with the other cores on the same chip. Caches are coherent though it is unclear if future designs will remain cache-coherent because of the significant overhead [BBC+08]. Common network topologies for inter-core communication include crossbar, bus, and ring. The node typically implements NUMA architecture (Non-Uniform Memory Access), which maps different memory banks to different chips. In such a design, cores may have non-uniform access latencies to different regions of memory.

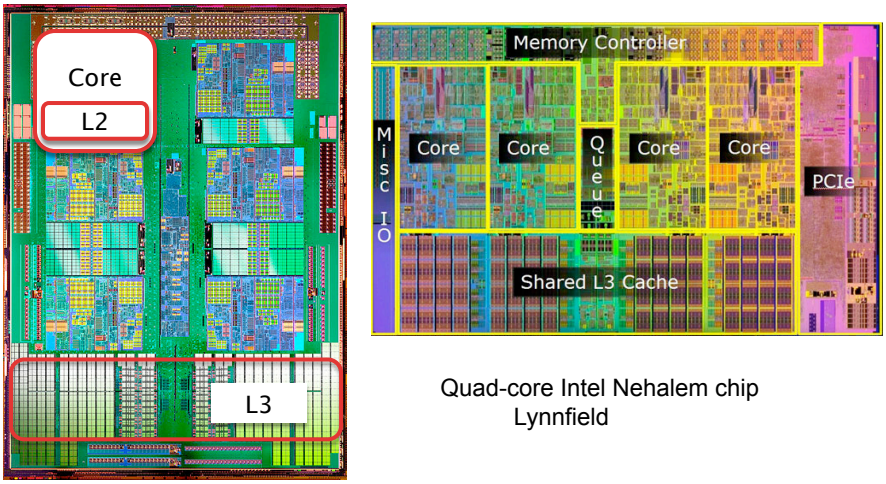


Figure 2.2: Two examples for 45 nm process technology

Fig 2.2 shows two examples of multicore processors available today. The first one is a hexa-core processor, code-named Istanbul from the AMD Opteron processor family. The 45 nm chip area includes 6MB shared L3 cache, 64KB private L1 and 512KB private L2 caches. The clock rate ranges from 2.2 to 2.8 GHz. This processor powers the Jaguar machine installed at Oak Ridge National Laboratory, which currently ranks 3rd in the Top500 list. The second chip (on the right) is a quad-core microprocessor from Intel. It is a Nehalem-based Xeon with 45 nm process technology, operating at 2.4-3.06 GHz. There is a 64KB private L1, 256KB private L2

and 8MB shared L3. Intel supports hyper-threading to improve parallelization on the chip by replicating certain parts of the processor [TEE+97]. The operating system can virtualize each physical core as two logical cores and schedule two processes simultaneously. For example, a chip containing four cores can scale to eight threads.

2.1.2 Massively Parallel Single Chip Processors

Massively parallel single chips provide a significant performance boost in node performance and can be used in supercomputers. Typically each core on a massively parallel chip lacks some of the ILP components but has a large number of ALUs (arithmetic logic units) compared to a CPU core. This minimizes the control complexity and the area on the chip, improving power consumption [CMHM10]. However, today's massively parallel single chip processors are specialized for certain application domains because they restrict the types of computation that can be performed. Unlike CPUs, massively parallel chips execute many threads concurrently but each thread executes very slowly. As a result, they may be attached to a general-purpose CPU which runs an operating system and serves as the controller or they might be general-purpose but not as powerful in single thread performance as a general-purpose multicore.

Unfortunately, having more ALUs does not translate into a proportional increase in performance. Today's chips are able to perform arithmetic operations a lot faster than we can feed them with data. This creates a major obstacle to performance also known as the "memory wall" [WM95]. Massively parallel single chips address this problem by introducing a large register file and a software-controlled memory hierarchy together with high intra-chip bandwidth. Managing the on-chip memory, however, comes at the expense of added programming overhead.

Some of the massively parallel single chip processors available today are Tilera [Til, TKM+02], Clearspeed [Cle], Tensilica [KDS+11, DGM+10], FPGAs [CA07] and GPUs. Each of these is designed with its own objective and targeting a different market segment. We look into GPUs in depth because of their wide adoption in scientific computing. According to the Top500 supercomputing rankings released in November 2011 [Top], three out of the five fastest supercomputers contain GPUs as the horsepower.

2.1.3 Graphics Processing Units

A **Graphics Processing Unit** (GPU) is specialized for *stream processing* [KDK+01]—in which computations become a sequences of *kernels*, functions that run under the single instruction, multiple threads model. This highly data parallel structure of GPUs attracted inter-

est from the scientific computing community and started the era of *GPGPU –General Purpose Computing on GPUs* [DLD+03, FQKYS04]. GPUs provide significant performance improvement not only in terms of arithmetic throughput but also in memory bandwidth for data parallel applications.

GPUs are not general-purpose processors. They require a general-purpose CPU as a *host*, to run an operating system and serve as a controller. Even though future systems may treat data motion differently, in current GPU-based systems, the host and device have physically distinct memories. The programmer controls the data motion between the two at the software level. Integrated GPUs are available in the market but they are usually far less powerful than those on a dedicated card.

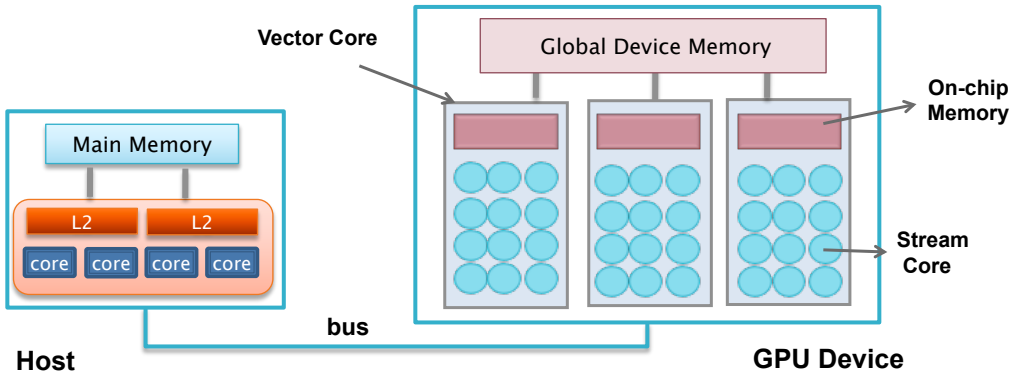


Figure 2.3: Abstract machine model of a GPU device connected to a host processor

Fig. 2.3 abstracts the computing unit and memory hierarchy in a GPU. A GPU device comprises groups of *vector cores*, each containing multiple *stream cores*. A vector core is capable of managing thousands of concurrent hardware threads. The stream cores, each of which is equipped with arithmetic logic units, are responsible for executing kernels. A stream core operates on an independent data stream but executes the same instruction with the other stream cores in the same vector core. A GPU kernel runs a virtualized set of scalar threads that are organized as a group of threads. The hardware dynamically assigns each *thread group* to a single vector core. Each thread in a thread group executes on a single stream core. GPUs further break down the thread group into subgroups. A subgroup executes the same instruction at the same time.

The hierarchy of computing units is reflected in the memory hierarchy as well. There is an off-chip *global device memory*, which has a high access latency and is accessible by all threads. There are two types of low latency on-chip memory; *private* and *shared memory*. The private memory is typically a set of registers, which is specific to a thread, and not visible to

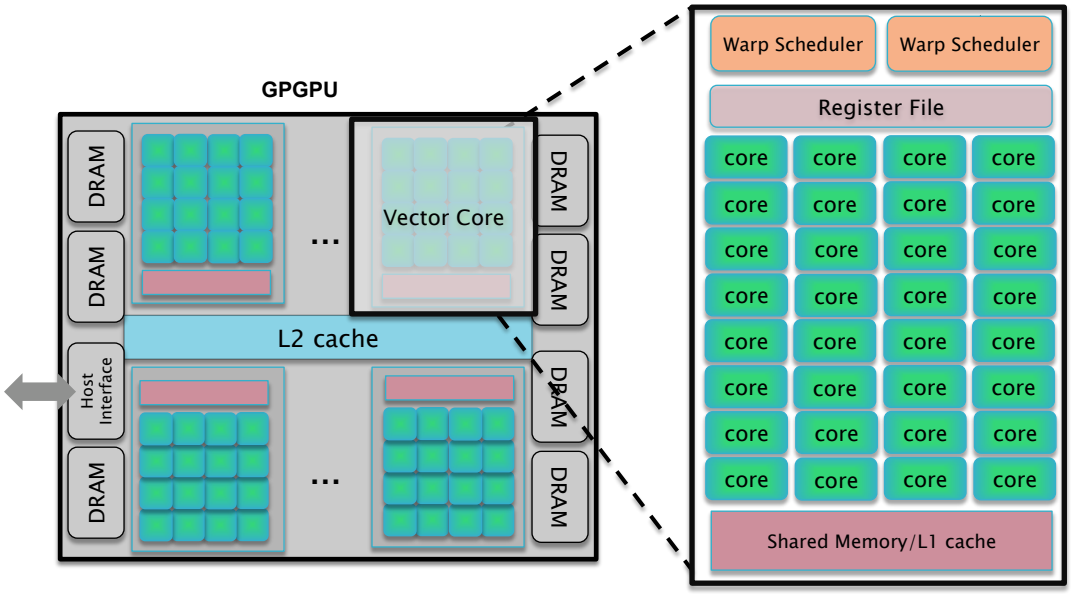


Figure 2.4: Nvidia GPU architecture

other threads. The shared memory is software and/or hardware-managed memory that is specific to a thread group, and accessible only by the threads belonging to that thread group.

The device can overlap computation with transfers to and from global memory. This is achieved by maximizing device *occupancy*, that is, the ratio of active threads to the maximum number of threads that a vector core supports. The common wisdom is that with sufficient occupancy, the processor effectively pipelines global memory accesses thereby masking their cost. The scarcity of on-chip memory constrains this goal, and the exact amount of realizable parallelism depends on the specific storage requirements of the kernel. The architecture requires the programmer find a good balance between the number of threads and their dataset size. A large number of threads have more potential to hide memory latency. But more threads mean fewer resources per thread. In addition to hiding transfer latency, it is important to avoid *thread divergence*. Threads within the same thread subgroup taking different branches cannot be executed in parallel on a vector core. They will be serialized.

The two largest dedicated graphics card designers, AMD and Nvidia, provide solutions for GPU-based High Performance Computing systems. Although both vendors have a similar device design, each vendor uses its own terminology for the computing and memory units. AMD refers to a vector core as a *compute unit* [AMD11] and Nvidia refers to it as a *streaming multiprocessor* [NBGS08]. Different GPUs have different numbers of vector cores as well as different

number of stream cores in a vector core. For example, the ATI Radeon HD 5870 GPU has 20 vector cores, each with 16 stream cores. The Nvidia Tesla 2050 has 14 vector cores, each with 32 stream cores.

The private memory on both vendors' GPUs consists of a large register file. The AMD GPUs initially offered only L1 cache as the shared memory, which is shared by a thread group. AMD Evergreen GPUs include software-controlled scratchpad memory, called *local data store* [AMD11]. Nvidia's shared memory, called *shared memory*, is managed at the software level. In addition to software-managed on-chip storage, in its latest GPUs, Nvidia provides L1 cache on a vector core. Thread-private data that does not fit into registers is placed in the global memory which is three orders of magnitude slower to access. This global memory region, for lack of a better term, is called *local memory* by Nvidia. Fig. 2.4 shows the architectural overview of an Nvidia GPU.

The two vendors differ in the naming of the thread and thread groups although the thread hierarchy is essentially the same. AMD refers to a thread as a *work item* and thread group as a *work group*. The work-items that are executed together, or the subgroup, is called a *wavefront*. Nvidia refers to a thread as *CUDA thread*, (or simply thread) and thread group as a *thread block*. The collection of CUDA threads, or the subgroup, that execute the same instruction at the same time is called a *warp*. The latest generation of Nvidia GPUs provides two warp schedulers, which are capable of issuing two different instructions to the same vector core.

In this thesis we perform our experiments on the Nvidia GPUs and will adopt the Nvidia terminology for the computing and memory units. We leave the detailed description about the GPU testbeds employed in our performance results to Chapter 6.

2.2 Application Characteristics

Although it may seem that applications have various distinct characteristics, the underlying computational methods exhibit common patterns of computation and data movement. Phil Colella [Col04] identified seven patterns, called "Seven Dwarfs", in scientific computing. The Berkeley View [ABC$^+$06] extended the 7 dwarfs to 13 "motifs" to characterize other patterns in and beyond scientific computing. Researchers have been studying the optimizations of the dwarfs over several years because the dwarfs are less complex to analyze than a whole application but representative enough to exhibit the same dependency pattern. Another merit of the classification is that these dwarfs can be used to benchmark computer architectures, programming models, and tools designed for scientific computing.

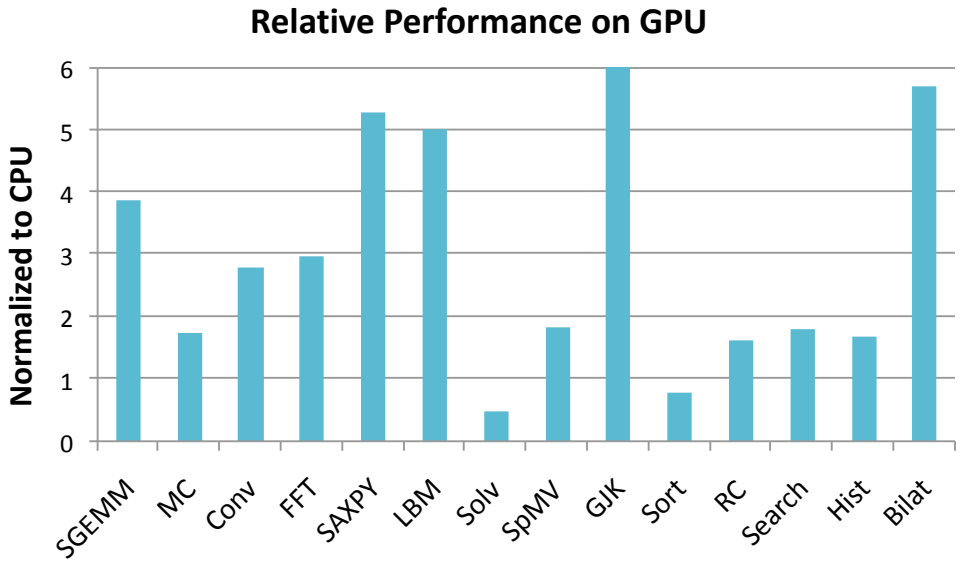


Figure 2.5: Comparison between Core i7 and GTX 280 performance on various applications. The data was collected from Victor Lee et. al [LKC+10].

The initial seven dwarfs are widely used in scientific computing. They are Dense Linear Algebra, Sparse Linear Algebra, Spectral Methods, N-body Methods, Structured Grids, Unstructured Grids, and MapReduce. Each shows a different level of sensitivity to memory performance. The dense matrix problems are typically computationally bound because they exhibit a regular memory access pattern and high data reuse. N-body problems are computationally expensive and may or may not exhibit regular memory accesses. Sparse linear algebra and structured grid problems are usually memory bandwidth limited and have very low reuse of data in the cache. Structured grids tend to have strided memory accesses. In contrast, spectral methods (e.g. FFT) and unstructured grid problems exhibit low spatial locality and unpredictable access patterns. They tend to be memory latency limited.

When a new architecture emerges, the motifs are evaluated on it to discover the potential of the architecture. For example, the Cell Processor [KDH+05a] and GPUs were evaluated [VD08, DMV+08, WSO+06, WOV+07] in terms of their performance and applicability in scientific computing. Lee et. al [LKC+10] studied so called “throughput computing kernels" on a traditional multicore CPU and a GPU to gain insight into the architectural differences affecting performance. Fig. 2.5 shows the performance results and Table 2.1 lists the characteristics of 14 kernels employed in the study. The experiments were conducted on an Intel quad-core Core

Table 2.1: Characteristics of the benchmarks and speedups of the GTX 280 over the Core i7.

Benchmark	Speedup	Berkeley Motif	Characteristics
SGEMM	3.9	Dense Linear Algebra	Compute-bound
Monte Carlo (MC)	1.8	MapReduce	Compute-bound
Convolution (Conv)	2.8	Structured Grids	Compute/BW-bound
FFT	3.0	Spectral Methods	Latency-bound
SAXPY	5.3	Dense Linear Algebra	BW-bound
LBM	5.0	Structured Grids	BW-bound
Constraint Solver (Solv)	0.5	Finite State Machine	Synchronization-bound
SpmV	1.9	Sparse Linear Algebra	BW-bound
Collision Detection (GJK)	15.2	Graph Traversal	Compute-bound
Radix Sort	0.8	MapReduce	Compute-bound
Ray Casting (RC)	1.6	Graph Traversal	BW-bound
Search	1.8	Graph Traversal	Compute/BW-bound
Histogram(Hist)	1.7	MapReduce	Synchronization-bound
Bilateral Filter(Bilat)	5.7	Structured Grids	Compute-bound

i7-960 processor (using all 4 cores) and Nvidia GTX 280 Graphics card. The peak single precision (SP) performance is 102.4 Gflops on the Core i7. The peak SP performance for the GTX 280 is 311.1 and increases to 622.2 by including fused-multiply-add. The peak double precision performance is 51.2 and 77.8 Gflops on the Core i7 and GTX 280, respectively. The Core i7 provides a peak memory bandwidth of 32 GB/s. The GTX 280 provides 4.7 times the bandwidth at 141 GB/s.

It is not surprising that the compute-bound kernels (*SGEMM, MC, Conv, FFT, Bilat*) exploit the large number of ALUs on the GPU. *SGEMM, Conv* and *FFT* realize speedups in the 2.8-3.9X range. The results are in line with the ratio between the two processor's single precision flop ratio. Depending on to what extent kernels utilize fused-multiply-adds, the flop ratio of the GTX 280 to Core i7 varies from 3.0 to 6.0. *MC* uses double precision (DP) arithmetic, and hence the performance improvement is only 1.8X, which is close to the 1.5X DP flop ratio between the processors. *Bilat* takes advantage of the fast transcendental operations on GPUs, resulting in 5.7X speedup over the Core i7. Even though the Radix *Sort* kernel is compute-bound, the implementation requires performing many scalar operations to reorder the data. Thus inefficient use of SIMD hurts the GPU performance.

The bandwidth-limited kernels (*SAXPY, LBM, and SpmV*) benefit from the increased

bandwidth in the GPU which has 4.7X times more peak memory bandwidth than Core i7. The speedups for the *SAXPY* and *LBM* are 5.3X and 5X, respectively. The 1.9 speedup for the *SpmV* kernel is modest because of the small on-chip storage (16KB) on the GPU. The CPU implementation takes advantage of the cache hierarchy for the input vector.

The *Solv* kernel is a rigid-body physics problem that simulates response to colliding objects. Due to the lack of fast synchronization operations on GPU, performance is worse than that on the Core i7. Another kernel where the barrier overhead is dominant is the *histogram* because of the reduction operation. *GJK* is a commonly used collision detection algorithm which requires lookups and a gather operation. The GPU implementation uses texture mapping units for lookups and its hardware supports efficient gather operations. Both contribute to its high performance over the Core i7. The *RC* kernel accesses non-contiguous memory locations and its SIMDization is not efficient. The GPU performance results in slightly better performance than the Core i7 (1.6X).

Overall, only two of the kernels *Sort* and *Solv* perform better on Core i7. *Sort* performs worse due to the lack of cache on the GTX 280. The current GPUs come with an L1 cache, which may change this fact. The *Solv* kernel's execution time is dominated by the synchronization overhead. The GPUs do not provide a global barrier mechanism entirely on the device and this fact is unlikely to change in the near future. The proposed software methods [VD08] do not ensure memory consistency across vector cores on the GPU. The kernels (*RC, Sort*) that can not efficiently utilize the SIMD units realize only a modest performance improvement.

In conclusion, GPUs offer the potential to improve performance over CPUs for applications that demand high memory bandwidth and computational power. Applications that require support for fast synchronization or irregular memory accesses realize a modest or no performance improvement on GPUs. One of the highly used motifs in scientific computing is structured grid problems, which tend to be bandwidth-limited. These problems exhibit performance improvements when ported to a GPU architecture. However, an outstanding difficulty is that programming on GPUs requires nontrivial knowledge of the architecture. We provide a programming solution to free the programmers from some of the burdens of developping their GPU applications using structured grids. The accompanying compiler to our programming model targets this class of applications. In the next section, we discuss the characteristics of structured grid problems in depth.

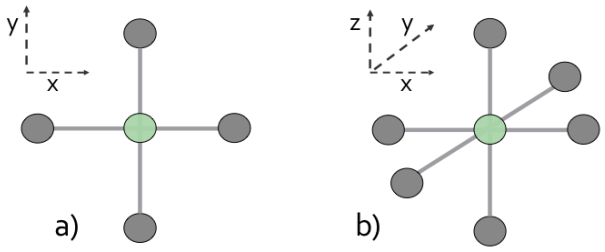


Figure 2.6: a) 5-point stencil b) 7-point stencil

2.2.1 Structured Grids

Structured grid problems appear in many scientific and engineering applications. They arise in approximating derivatives numerically, and are used in a variety of finite difference methods for solving ordinary and partial differential equations [Str04]. They are also prevalent in image processing. Some of the application areas include physical simulations such as turbulence flow, seismic wave propagation, or multimedia applications such as image smoothing.

The structured grid computation involves a central point and a subset of neighboring points in space and time, all arranged on a structured mesh. The weight of the contribution of a neighbor may be the same (constant coefficient) or may vary (variable coefficient) across space and/or time. We use *stencil* to refer to the neighborhood in the spatial domain. The stencil operator applied to each point is the same across the grid. The computation is implemented as nested for-loops which sweep over the grid and update every point typically in place or between two versions of the grid [2].

The two most well known stencils are the 5-point stencil approximation of the 2D Laplacian operator and the corresponding 7-point stencil in 3D, both shown in Fig. 2.6. As a motivating example, we consider the 3D heat equation $\partial u/\partial t = \kappa\nabla^2 u$, where ∇^2 is the Laplacian operator, and we assume a constant heat conduction coefficient κ and no heat sources. We use the following explicit finite difference scheme to approximate derivatives and solve the problem on a uniform mesh of points.

$$\frac{u^{n+1}_{i,j,k} - u^{n}_{i,j,k}}{\Delta t} = \frac{\kappa}{\Delta x^2}\left(u^{n}_{i,j,k-1} + u^{n}_{i,j-1,k} + u^{n}_{i-1,j,k} - 6u^{n}_{i,j,k} + u^{n}_{i+1,j,k} + u^{n}_{i,j+1,k} + u^{n}_{i,j,k+1}\right).$$

The superscript n denotes the discrete time step number (an iteration), the triple-subscript i, j, k denotes the spatial index. For simplicity, we assume equal spacing Δx of the mesh points in all directions, and equal spacing of timesteps t_n at a fixed interval of $\Delta t = t_{n+1} - t_n$. Note that the

[2] Sweep can involve three versions of the grid such as solving a wave equation by fully-explicit finite difference.

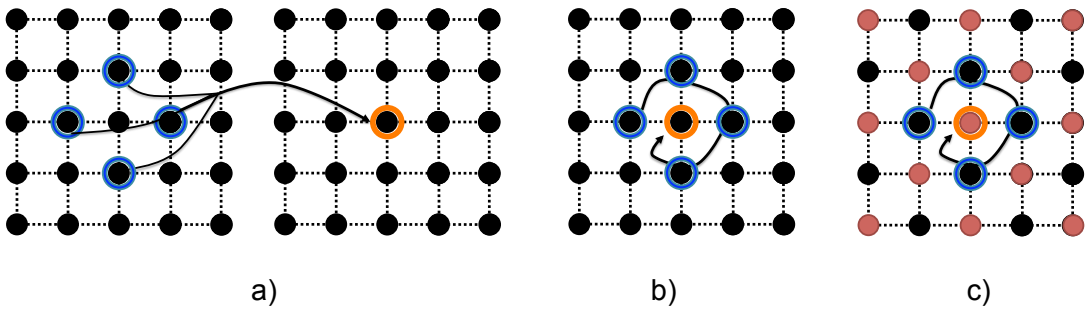


Figure 2.7: Iterative Methods for Solving Linear Systems: a) Jacobi Method b) Gauss-Seidel Method c) Gauss-Seidel Red-Black Method

above formula is a 7-point computational stencil applicable only to inner grid points, and for simplicity we have omitted the treatment of boundary points.

Iterative Methods for Solving Linear Systems

Many stencil methods are iterative; they sweep the mesh repeatedly until the system achieves sufficient accuracy or reaches a certain number of timesteps. One of the iterative methods is Jacobi's method [BS97] which uses two copies of the grid; one for the values under construction and one for the values from the previous timestep, as shown in Fig 2.7-a. The use of two grids makes Jacobi's method highly parallelizable. Any point in the output grid can be computed independently from any other point in the output grid. However, the method requires more storage and uses more bandwidth than the Gauss-Seidel method [JJ88], which performs sweeps in place. As depicted in Fig 2.7-b, the write grid is also the read grid. In this method, some of the values of neighbors are old (from the previous step), and some of them are from the current step. However, the data dependency in the read/write operations makes the method hard to parallelize. Gauss-Seidel Red-Black [WKKR99] solves the dependency problem by updating every other point and converges twice as fast as the Jacobi method. As shown in Fig 2.7-c, the red points are updated first using black points as input and then in the second sweep the black points are updated by using the red points as input.

Table 2.2 shows the fully-explicit time stepping on the 3D heat equation. We maintain two copies of the grid; `Unew` and `U` and swap the pointers at the end of each sweep. In this naive implementation the kernel performs 7 loads and 1 store in order to update a single point in `Unew`.

We have discussed the characteristics of the structured grid problems and will continue our discussion about their parallelization and optimization in Chapter 4. Chapter 6 will provide

Table 2.2: Code for the 3D heat equation with fully-explicit finite differencing. `Unew` corresponds to u^{n+1} and `U` to u^n and $c_0{=}1-6\kappa\Delta t/\Delta x^2$ and $c_1{=}\kappa\Delta t/\Delta x^2$.

```
1  while( t++ < T ){
2
3    for (int z=1; z<= k; z++)
4     for (int y=1; y<= m; y++)
5      for (int x=1; x<= n; x++)
6       Unew[z][y][x] = c0 * U[z][y][x] + c1 * (U[z][y][x-1] + U[z][y][x+1] +
7                         U[z][y-1][x] + U[z][y+1][x] + U[z-1][y][x] + U[z+1][y][x]);
8    double*** tmp;
9    tmp = U; U = Unew; Unew = tmp;
10 }//end of while
```

performance results of commonly used stencil kernels in scientific and engineering applications. The chapter will compare the performance of the Mint-generated kernels with their manual implementations.

2.3 Parallel Programming Models

The performance of an application running on a traditional multicore processor highly depends on the use of multi-threading, SIMDization, and exploitation of on-chip memory. Future systems that rely on massive parallelism on a chip will only exaggerate today's software challenges. Application developers face three main challenges: 1) Extract sufficient parallelism from the application. 2) Manage the memory hierarchy for data locality. 3) Limit the changes that have to be made to the existing codes. These challenges require significant programming effort to realize performance expectations. In this context, appropriate software tools can help the programers adopt new technologies because they abstract the underlying parallel architecture and facilitate the programmer's view of the computing units and memory hierarchy.

The most successful efforts at coping with architectural change have relied on application libraries (such as LAPACK [ABB+99] and PETSc [BGMS97]) or custom domain-specific languages [Mat] that are focused on the mathematics and thus hide all implementation details. We present a different approach–domain specific source code transformations, assisted with programmer annotation and compiler options. This thesis demonstrates Mint, a programming model and translator based on our approach, that targets stencil methods, running on the massively parallel single chip processor: GPUs.

A programming model is a layer between the application and the architecture. A high level programming model puts the ease of use front but can penalize performance. It doesn't allow the programmer to control architectural details that might impact performance. A low-level model provides more expressiveness such as thread and data management, resulting in high performance but it hinders productivity of the programmer. We contend that to meet both productivity and performance goals we need to restrict the application space so that we can incorporate semantic content into the translation process. This thesis embodies a domain-specific approach for stencil methods. We enable the programmer to control the hardware at a much higher level. In exchange, we generate code customized to the application avoiding the cost of generalizing assumptions made by conventional compiler and language constructs. As a result, application developers use their valuable time to focus on the application, rather than on learning the idiosyncratic features of the hardware, yet still enjoy the improved performance compared to a general-purpose approach.

Although a large number of programming models have been proposed for GPUs as libraries [Nvi07, ADD+09], and language extensions [NBGS08, BB09], only a few have gained traction within the parallel computing community. In the next section, we provide the background in related work starting with OpenMP [CJvdP07] which is one of the most widely accepted models to program general-purpose multicores. It hasn't been extended to support massively parallel single chip processors but it has inspired the design of others such as PGI Accelerator model [Wol10b] and OpenMPC [LME09]. We also discuss CUDA [NBGS08], an API developed by Nvidia to program Nvidia GPUs. It constitutes the upper limit for the achievable performance by a programming model designed for GPUs since it is sufficiently low level. After CUDA, we briefly present the OpenCL effort. We will also discuss general-purpose annotation-based programming models and domain-specific source-to-source translators targeting GPUs.

2.3.1 OpenMP

OpenMP [CJvdP07] consists of compiler directives and library routines for shared memory parallel programming. In a shared memory model, a collection of threads share a single address space and communicate through shared variables. OpenMP is a directive-based high level implementation of shared memory model and an OpenMP-capable compiler parallelizes the code based on the annotations provided by the programmer. OpenMP uses a fork-join execution model but the thread identity is invisible to the programmer because the compiler optimizes away most of the thread creation and termination. The program starts with a single thread of ex-

ecution and spawns new threads when it encounters a region annotated as *parallel*. Loop-based parallelism is the most common means of parallelization in OpenMP. The programmer annotates the parallelizable loops and the compiler divides the iteration space into chunks and assigns these chunks to threads. Each thread concurrently executes the loop body but works only on its own chunks. OpenMP supports task parallelism as well by allowing different threads to execute independent code sections.

OpenMP is considered to be a convenient way to parallelize an application. Since the compiler directives may be ignored, OpenMP allows a unified base for both serial and parallel applications. However, OpenMP's high level model takes away the programmer's control over parallelism, data distribution and thread affinity, resulting in scalability problems [Yel08]. Some hardware vendors (e.g. SGI) support highly tuned OpenMP libraries with additional features to control data layout [Pop].

2.3.2 CUDA

CUDA [NBGS08] is an API specifically designed for programming Nvidia GPUs to execute programs written with C, C++ and Fortran[3]. CUDA is very tightly coupled with the Nvidia GPUs. The programming constructs directly map to the thread and memory hierarchy of the GPU device. The user defines functions called *kernels* that will be off-loaded to the GPU. The kernel functions are declared as `__global__` to distinguish them from other functions in the program. A call to one of the kernel functions generates thousands of threads on the vector cores and the function is executed once for each thread in parallel. A kernel contains the code performed by a single thread. Each thread uses its ID to identify its work assignment. The programmer organizes kernel threads into *blocks* and blocks into *grids* as shown in Figure 2.8. A thread block is a set of concurrently executing threads which can communicate through on-chip memory. When invoking a kernel, the number of threads per block and the number of blocks per grid are specified. Typical block sizes form powers of two. The number of blocks per grid varies according to the input size.

For efficiency reasons, thread scheduling and management are implemented in hardware. Hence, task creation, scheduling and context switch have almost zero overhead at runtime. The hardware manages threads in groups of 32, called *warps*. For full efficiency, the programmer should organize threads in a warp in a way that they follow the same execution path. A warp serializes the execution of divergent paths, thus reducing concurrency. The threads in a block

[3]Third party wrappers for CUDA are available for Python, Perl, Java, Ruby, Lua, and MATLAB.

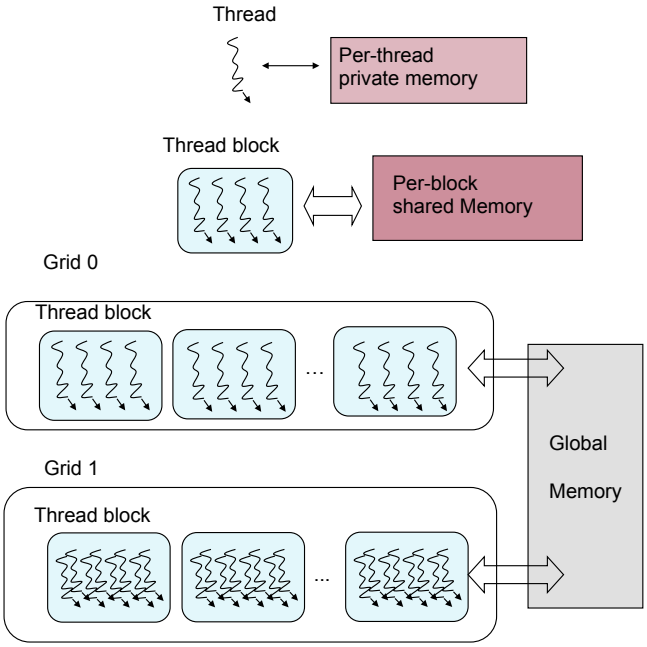


Figure 2.8: Memory and thread hierarchy in CUDA

are scheduled to execute on the same vector core and run to completion. CUDA virtualizes the thread block execution by allowing more blocks than available vector cores. As a result, CUDA blocks logically run concurrently. But this implies that any inter-block dependency should be avoided since the block execution order may vary.

Within a block, threads synchronize at a barrier and communicate through shared variables residing in the *per block shared memory* as shown in Figure 2.8. Shared memory variables are distinguished syntactically in a program with `__shared__` keyword. In addition, each thread has its own private memory, typically consisting of registers. Lastly, global memory is visible by all threads of a kernel, and is located on the GPU's DRAM. The system memory on the host is not directly accessible by the stream cores. The application must allocate some space in global memory and use this space to copy the data between the host and the GPU.

When designing a kernel, the performance programmer should carefully determine which variables are placed in the register file, on-chip shared memory, or global memory. Global memory is larger than shared memory but has a higher access latency. The register file is the fastest level of memory on the chip, but only a group of registers is dedicated to a single thread. As a result, the number of threads and the number of registers per thread are inversely proportional. Tuning these parameters greatly affects the performance.

2.3.3 OpenCL

OpenCL [Opeb], managed by the Khronos Group [The11], is an effort with the objective of having an open industry standard to embrace Intel, AMD, Nvidia, and ARM architectures. It aims to provide a uniform and portable application code for a diverse set of architectures including multicore CPUs, GPUs and Cell BE [KDH+05a]. Currently, OpenCL's performance lags behind CUDA on the Nvidia GPUs. According to a comprehensive performance study [FVS11], CUDA outperforms at most 30% better than OpenCL on the Nvidia GPUs. There are two main reasons that contribute to the performance gap. First, CUDA is specifically tailored towards the Nvidia GPUs, and hence, it can better utilize the hardware. OpenCL compromises performance for the sake of portability. Second, CUDA is more mature and provides more compiler optimizations. OpenCL is immature but gaining more attention from the vendors. It is likely that OpenCL will be a good alternative to CUDA in near future. AMD eagerly supports the development of OpenCL and recently released AMD Accelerated Parallel Processing Software Development Kit for its ATI video cards [AMD11].

2.3.4 Annotation-based Models

OpenMPC is a general-purpose programming model that supports an extended OpenMP syntax for GPUs [LME09, LE10]. Its compiler generates many optimization variants of the same input code, and the user guides optimization through a performance tuning system. The OpenMPC framework comes with a search space pruner for the applicable optimizations to reduce the optimization space. However, the number of tuning configurations after pruning is still in the order of 100s.

OpenMPC maps a non-hierarchical programming model (OpenMP) to to a hierarchical programming model (CUDA), resulting in fundamental limitations. Notably, it cannot expose the memory hierarchy and multiple types of parallelism in CUDA to the programmer. OpenMPC only parallelizes the outermost loop of a loop nest whereas our Mint model parallelizes an entire loop nest. As a result, Mint can generate multi-dimensional CUDA thread blocks. This capability is particularly essential for 3D problems; one level of parallelism is not adequate to occupy the device effectively in finite difference stencils, and multidimensional partitioning is required. The results in Chapter 6 show a significant benefit from parallelizing all levels of a loop nest in three dimensions. Moreover, because on-chip memory optimizations play an important role in performance, Mint heavily invests in shared memory and register optimizations. By comparison, OpenMPC uses shared memory for scalar variables only and cannot buffer arrays in shared

memory.

We compared the performance of the code generated by OpenMPC with the code generated by Mint. For a 2D kernel implementing the 5-point heat solver, Mint realized 14.4 Gflops while OpenMPC achieved 7.3 Gflops for an input size of 4Kx4K. In a 3D kernel implementing the 7-point heat solver, the performance gap is even larger. For an input size of 256^3, Mint realized 22.2 Gflops while performance dropped to 1.06 Gflops for OpenMPC. The low performance of OpenMPC stems from the fact that OpenMPC parallelizes only the outermost loop and does not create enough thread blocks to occupy the device. The performance penalty is more significant in 3D kernels because the size of the outermost loop is smaller than that of 2D.

The Portland Group[4] proposed the PGI Accelerator model [Wol10b], which is a collection of compiler directives to specify regions of the code for acceleration. The Portland Group developed commercial compilers for C and Fortran that implement the model. Similar to Mint, the PGI compiler uses shared memory and multi-dimensional thread blocks. Unlike Mint, PGI's approach is intended to be general-purpose. As we show in Chapter 6 and 7, the payoff for Mint is improved performance for the selected application domain.

We compared the performance of the codes generated by the PGI compiler and the Mint compiler. For the Heat 7-pt kernel, Mint realized 22.2 Gflops while the PGI compiler delivered about half the performance of Mint: 9.0 Gflops. The PGI version uses on-chip memory to improve data locality, but not as effectively as Mint. Mint uses registers in lieu of shared memory, reducing pressure on shared memory and thereby increasing device occupancy (See Section 5.6 for details). We tested both compilers with another stencil kernel (19-point), which requires the same amount of shared memory even registers are used in the optimization. For the 19-point kernel, Mint and PGI realized 15.8 and 11.3 Gflops, respectively. The Mint compiler implements sliding window optimization through the `chunksize` clause, improving reuse in stencil kernels (discussed in Section 5.8). In short, we use domain-specific knowledge to deliver higher performance.

The Hybrid Multicore Parallel Programming (HMPP) [BB09] from CAPS entreprise[5] offers a directive-based model to program hardware accelerators. While a Mint user annotates for loops for acceleration, a HMPP user annotates *codelets*, functions that can be offloaded to the device. The model restricts what can go into a codelet. For example, it can not return a value or access any global variable. Another directive-based language is HiCuda [HA09], which supports the same programming paradigm already familiar to GPU users because it exposes the details of

[4] http://www.pgroup.com/

[5] http://www.caps-entreprise.com/

the GPU hardware to the programmer through annotations. For example, its compiler can cache global memory data in shared memory, but the shared data and its size as well as synchronization points have to be specified through directives. Mint requires only shared compiler flag to be switched on, then it automatically handles which data to be placed in shared memory and inserts the synchronization points.

The OpenACC Application Program Interface [Int11] is a new programming standard developed by PGI, Cray, Nvidia and CAPS for GPGPU computing. It is a directive-based model and influenced by the PGI Accelerator Model with some additions from Cray's accelerator model. The user indicates the regions of a program to be offloaded to an accelerator device. The rest of the program executes on the host CPU. The interface includes compiler directives, library routines and environment variables. There are a large number of directives to support data transfers between the system memory and device memory. The interface exhibits some differences in its support for C/C++ and Fortran mainy because how arrays are treated under Fortran. PGI, Cray and CAPS are expected to provide OpenACC API-enabled compilers and runtime systems in the 1st quarter of 2012.

2.3.5 Domain-Specific Approaches

Regarding domain-specific source-to-source translators, there have been several models proposed to facilitate CUDA programming. Chafi et al. proposed the OptiML language for machine learning for GPU parallelism and a system called Delite to design other domain-specific languages [CSB+11]. Lionetti et al. [LMB10] implemented a domain-specific translator with a Python interface for a cardiac simulation framework, which solved a system of ODEs in reaction diffusion equations. Christen et al. [COS11] developed a translator called PATUS that takes a description of a stencil computation along with a description of an optimization strategy. The code generator can produce CUDA or OpenMP source codes. Although PATUS has the potential to express more general optimizations than Mint, we found that optimizations described by the authors are currently accomodated by Mint's optimizer and controlled at a high level through programmer annotations. Thus Mint brings less overhead to the programmer.

Another work on stencil computation for GPUs is OpenCurrent [opea]. OpenCurrent is an open source C++ library from NVIDIA for solving PDEs with structured grids. The library is composed of 3 main objects: Grids, Solvers and Equations as computational building blocks and has its own data structure. However, the solvers in OpenCurrent are not optimized and the library has limited functionality.

2.4 Summary

In this chapter, we discussed trends in computer architecture and the potentials of massively parallel single chip processors to address the performance/watt problem. As we mentioned previously, we expect that exascale machines will rely on massive on-chip parallelism which has to be managed in software. However, applications present a daunting array of optimization strategies. The programmers have to wade through a sea of possible optimizations to arrive at a combination that delivers optimal performance on a given architecture. The result is that programmers often resort to trial and error.

To address the programming issue of massively parallel chips, we propose the Mint programming model. Mint allows the programmer to express parallelism at a high level. Its compiler parallelizes loop-nests, performs data locality optimizations and relives the programmer of a variety of tedious tasks. Mint's optimizer is not general-purpose. It targets stencil-based codes that appear in Structured Grid problems. By restricting the application domain, we are able to achieve good performance on massively parallel single chip processors. In the following chapters, we introduce the interface of Mint Programming Model, its translator and optimizer, performance results on commonly-used stencil kernels as well as on real-world applications.

Chapter 3

Mint Programming Model

This chapter introduces the Mint Programming Model designed for stencil computations to facilitate programming on a system equipped with a massively parallel single chip processor. The Mint model has two primary objectives: 1) to increase the productivity of the programmer, and 2) to provide competitive performance with hand-coding. Mint assists the programmer with an intuitive mapping from the high level interface to low-level hardware specific optimizations. The model is based on compiler directives which enable programmers to incrementally migrate their applications to a massively parallel processor using standard C. The programming interface includes compiler options to improve performance further.

Before we introduce the interface of the model, we discuss our assumptions about the underlying hardware system. We continue with the Mint execution and memory model. Next, we present the details of each directive and provide a simple program to illustrate the purpose of the directives. Lastly, we briefly describe the compiler options but leave the detailed discussion to Chapter 5 because they are tightly coupled to the target architecture. This chapter can serve as a user manual for a novice Mint programmer.

3.1 System Assumptions

The Mint programming model assumes a system design depicted in Fig. 3.1, comprising a traditional multicore processor and a massively parallel single chip processor. To emphasize the functionality of the two processors, we refer to the traditional multicore as the *host* and the massively parallel processor as the *accelerator*. Since massively parallel chip technology is in a state of flux, our model abstracts away some aspects of how the system functions.

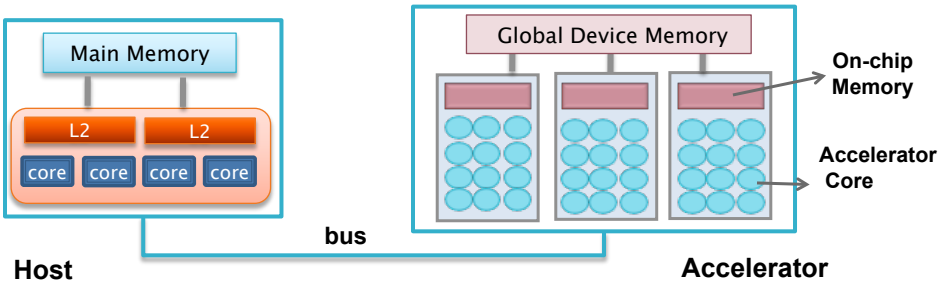


Figure 3.1: Abstract Machine Model viewed by the Mint Programming Model

The first abstraction concerns address spaces. We assume that the host and accelerator have physically distinct memories and the host controls all data motion between the two. However, Mint is neutral about how the data motion is brought about. Future systems may treat data motion differently, for example, the accelerator may be able to initiate data transfers. Alternatively, the accelerator might be integrated with the host CPU in the same package and share main memory with the CPU. Mint assumes the accelerator and host have separate address spaces and it is costly to move data in between. Hence, the programmer should avoid data transfers as much as possible.

The second abstraction concerns the computing capabilities. The accelerator cores are specialized to perform data parallel operations faster than the host cores, but are incapable of serving as a general-purpose processor. As a result, an application may benefit from using the host and the accelerator in different phases of an application. The host invokes multithreaded functions on the accelerator, which are execute as a sequence of long vector operations. These operations are hierarchically organized in two levels: First, they are partitioned into groups by the accelerator and assigned to different groups of cores. Second, the work of each group is further partitioned into pieces and assigned to individual cores. The data elements are computed independently in an undefined order.

The final abstraction concerns to the memory hierarchy of the accelerator. Mint assumes that there is fast and slow memory on the accelerator. Slow memory is the device global memory which is visible to all the cores on the accelerator. Fast memory is on the chip and accessible by only a group of cores. On-chip memory may comprise of a large register file, software-managed memory and/or hardware-managed cache. The fast memory delivers much higher bandwidth and a lower latency than device memory.

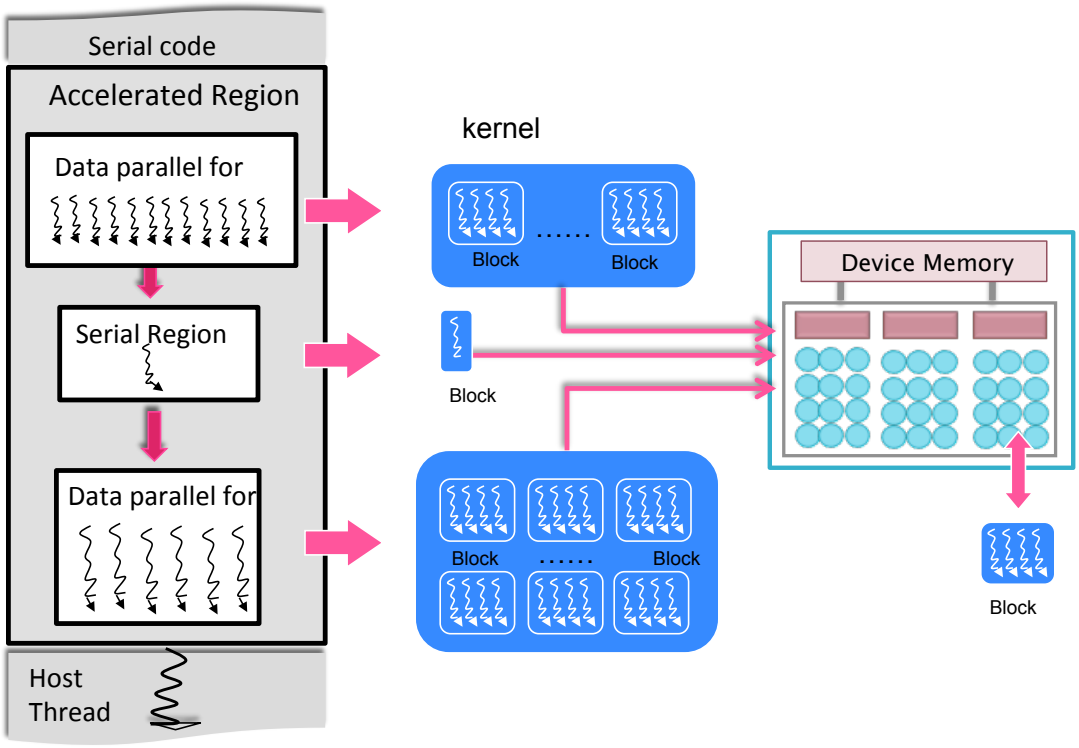


Figure 3.2: Mint Execution Model

3.2 The Model

A Mint program is a legal C program, annotated with Mint directives. These annotations serve to inform the compiler only, as Mint provides no executable statements. The syntax for Mint annotations has some similarities to OpenMP [CJvdP07], but includes directives and clauses tailored for the system assumptions discussed in Section 3.1. The Mint interface provides 5 compiler directives (pragmas). Briefly these are:

1) `parallel` indicates the start of an accelerated region,

2) `for` marks the succeeding loop-nest for acceleration,

3) `barrier` specifies synchronization,

4) `single` handles serial sections within a parallel region,

5) `copy` expresses data transfers between the host and accelerator.

Before going into details of these directives, we present the Mint execution and memory model.

3.2.1 Execution Model

A Mint program contains one or more designated accelerated regions. Each of these regions contains code sections that will execute on the accelerator under the control of the host. Although, all code in the region is not able to run on the accelerator, rather, only *kernels* can. A kernel is typically an annotated work-sharing nested-loop, or infrequently annotated serial region. All other code that is not in the accelerated region runs on the host.

A host thread starts the execution of the Mint program and offloads the multithreaded kernels to the accelerator. There is an implicit synchronization point at the end of each kernel unless the programmer specified otherwise. When the accelerator threads complete the execution of the kernel, they terminate and the host thread resumes. The accelerator applies two levels of parallelism on the kernel. First level is coarse-grain, where kernel threads are organized into thread blocks to execute across group of cores in parallel. Second level is fine-grain, where threads in the same thread block concurrently execute on the cores belonging to the same group of cores. In the coarse level, there is no synchronization mechanism while in the second level, threads can synchronize.

3.2.2 Memory Model

Under Mint variables are assumed to live on the host. Inside of an accelerated region, variables temporarily live on the accelerator. Since the accelerator memory is physically separate from the host memory, all data movement between two memories must be initiated by the host through runtime library calls. In the Mint model, data movement is managed by the compiler with the assistance of the programmer annotations. However, Mint's ability to keep track of the underlying data motion is minimal. The programmer must be aware of the separate address spaces. For example, it is programmer's responsibility to ensure that data is transferred to the device memory at the entry of a parallel region and transferred back if needed to the host at the exit of a parallel region. Mint takes care of storage allocation and deallocation of the variables on the device. At the exit of a parallel region the allocated memory for the device variables will be freed and the content will be lost. As a result, threads created in different parallel regions cannot communicate through device memory. If a programmer needs to access device variables on the host inside a parallel region, let's say for IO, then she needs to explicitly insert copy primitives to the program. In such cases, Mint conserves the allocated space for variables and their content on the device memory.

In the Mint model, on-chip memory on the device is managed by the compiler with the

compiler options provided by the programmer. The programmer does not need to know how the on-chip memory works but should be aware of their trade-off. More on-chip memory usage means fewer concurrent threads running on the device. Excessive usage can be counterproductive and diminish performance. Moreover, the limited on-chip memory size may lead to compile or runtime errors.

3.3 The Mint Interface

Mint consists of a compact set of attributes for annotating a C program, which can then be automatically translated into efficient accelerator code. In C, Mint directives are specified using the `#pragma` mechanism followed by `mint` keyword. A compiler which does not support Mint will ignore the user annotations. We next describe the 5 Mint directives.

3.3.1 Parallel Region Directive

A parallel region is a structured block of code that will be executed on the accelerator. This region indicates a distinct address space and creates an accelerator scope for the variables. Before control enters a parallel region, any data used in the region must have previously been transferred using the `copy` directive. When the host thread reaches a `parallel` directive, it prepares the data on the accelerator by allocating space and performing the data transfers guided by the `copy` directive. It is programmer's responsibility to specify the variable names involved in the transfers, their shape and number of dimensions.

Mint does not support a parallel region nested within another parallel region. Logically this would mean that a superior accelerator is attached to a moderate accelerator which is attached to the host. In contrast, an OpenMP nested parallel region creates a hierarchy of threads: the master thread spawns threads each of which can spawn more threads as they encounter parallel region, creating a tree structure. In our model, accelerator threads cannot spawn other accelerator threads. As a result, they can only appear at the leaves of the thread hierarchy. In the future if needed, Mint can support such nested parallelism by generating host threads in the higher levels and accelerator threads in the lowest level of the hierarchy.

In the current design, a parallel region cannot span multiple code files or multiple routines. Hence, program cannot branch into or out of a parallel region. Future work, inspired by the orphaned for-loops [CJvdP07] in OpenMP, will allow branching out of a parallel region to support function calls.

3.3.2 For-loop Directive

The `for` directive marks the succeeding `for` loop (or nested loops) for acceleration and manages data decomposition and work assignment. This work-sharing construct launches light-weighted threads to execute the enclosed code region as a kernel. There is no implied barrier at the entry to the region, however, there is an implied barrier at the end to synchronize the accelerator threads with the host thread. If there is `nowait` clause attached to a `for` loop directive, the host thread will resume execution and can perform some local work. Otherwise, it will wait until the accelerator threads join. The `for` loop directive should be enclosed statically within a parallel region in order to be accelerated. The scope of the directive can not span multiple code files. or multiple routines.

Mint parallelizes a loop nest by associating one logical accelerator thread with some number of points in the iteration space of the nest. It then partitions and maps the logical threads onto physical ones, guided by three clauses that the programmer employs to tune the loop. These clauses are:

- **nest (# | all)** indicates the depth of `for`-loop parallelization within a loop nest, which can be a constant integer, or the keyword `all` to indicate that all the loops are independent, and hence parallelizable. If the `nest` clause is not specified, Mint assumes that only the outermost `for` loop is parallelizable. Depending on the value of the clause, Mint can create multi-dimensional thread blocks to parallelize the specified loop nest. Multi-dimensional thread geometry does not flatten the iteration space across loops, preserving the data locality. The threads may have neighbors in higher dimensions as opposed to having only left and right neighbors.

- **tile (** t_x, t_y, t_z**)** specifies how the iteration space of a loop nest is to be subdivided into *tiles*. A data tile is assigned to a group of threads and the sizes are passed as parameters to the clause. In practice, a tile corresponds to the number of data points computed by a thread block.

- **chunksize (** c_x, c_y, c_z**)** aggregates logical threads into a single thread. Each thread serially executes these logical threads within a serial `for` loop. Chunking enables the programmer to manage the granularity of the workload mapping to threads within a thread block. Together with the `tile` clause, the `chunksize` clause establishes the number of threads that execute a tile. Specifically, the size of a thread block is $threads(t_x/c_x, t_y/c_y, t_z/c_z)$, as

depicted in Fig. 3.3. In the absence of `tile` and `chunksize` clauses, the compiler will choose default values[1].

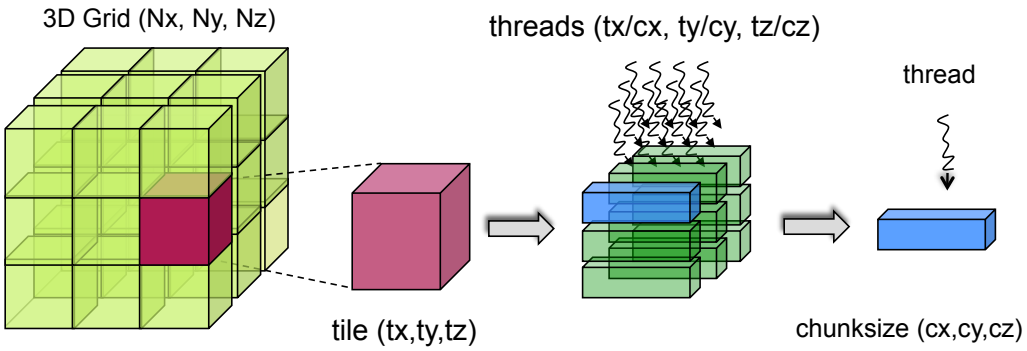


Figure 3.3: A 3D grid is broken into 3D tiles based on the tile clause. A thread block computes a tile. Elements in a tile are divided among a thread block based on chunksize clause. Each thread computes chunksize many elements in a tile.

3.3.3 Data Transfer Directive

The data transfer directive expresses data transfers between the host and device memory. We choose to get help from the programmer in terms of *when* the copy should occur, and *which* arrays take part. However, we made its interface easy so that it exposes minimal details of the underlying architecture. The syntax for the directive is as follows:

`#pragma mint copy(src|dst, toDevice|fromDevice, [`N_x`, `N_y`, `N_z`,...]).`

The source (or the destination) variable is the name of the host variable and should be declared before the `copy` directive is used. The parameter *toDevice* (or *fromDevice*) indicates the direction of the copy. The remaining parameters specify the array dimensions from fastest to slowest varying dimension.

Even though variables are declared and reside on the host, they temporarily live on the accelerator for the duration of a `parallel` region. Thus, the lifetime of transferred data is limited to the `parallel` region the transfer belongs. Mint temporarily binds a host array to the corresponding accelerator array for a `parallel` region: it handles the declaration of the accelerator array, allocation of it on the accelerator, and data transfers between two memory spaces. Mint unbinds them upon region exit (frees up storage on device memory upon exit of a

[1] Throughout this dissertation, we use a default of $16\times16\times1$ tiles with a chunksize of 1 in all dimensions. However, the default is configurable.

`parallel` region). Scalars are handled differently than arrays. The `copy` directive should not be used for scalar variables because they are passed as parameters to the kernel and implicitly transferred over the device memory.

There are some restrictions where the `copy` directives can appear in the program. The transfers *toDevice* should be placed right before the `parallel` region they belong to with no statements in between. The transfers *fromDevice* should appear right after the `parallel` region they belong to with no statements in between. Both can appear inside a parallel region with no restriction. The transfers which do not follow these rules will be ignored by Mint with a notification message. A `copy` directive implicitly triggers to series of operations under Mint depending on where it occurs. We will discuss these operations in Section 4.2.1 in Chapter 4.

Mint checks if all the vector variables used by a `parallel` region are transferred over to the accelerator. If not, it will issue an error with an informative message. However, it cannot successfully infer the shape of an array to carry out the transfers without the programmer's help. While the `copy` directive does not describe how data motion will be carried out, it does expose the separation of host and accelerator address spaces. Even on platforms with separate address spaces we may be able to elide the data flow through program analysis. Such optimization would rely on inferring data motion from context by capturing array shape information from static C declarations, or from dynamic memory allocations. The required program analysis techniques remain as future work.

3.3.4 Other Directives

The `mint barrier` directive is a global barrier to synchronize the accelerator with the host and it guarantees that all changes made to device memory are visible to all device threads. This directive should be used to synchronize threads when the `nowait` clause is attached to `for` loop directives so that the host thread waits for completion of the accelerator execution.

The `mint single` directive indicates serial execution by only one accelerator thread. Serial sections are expensive since it results in poor occupancy in the accelerator. Most of the execution units will be idle, thus the directive should be used infrequently. Unlike OpenMP, this directive should not be used for I/O or pointer swap operations because these operations are performed on the host not by an accelerator thread.

3.3.5 Reduction Clause

The `reduction` clause is allowed on the `for` directive to perform a reduction operator on a scalar variable using the accelerator arrays. A private copy of the variable is created for each vector core and initialized for that operator. At the end of the reduction, the results for each vector core is combined on the host. The final result is written to the corresponding host variable. The format of the `reduction` clause:

```
#pragma mint for reduction(operator:var_name)
```

Currently, we only support $add(+)$ operator with initialization value of zero. Future work will allow other common operators such as $max, min, multiply(*)$ etc.

3.3.6 Task Parallelism under Mint

Mint does not directly support task parallelism in the sense of MIMD (multiple instruction and multiple data). However, it is possible to implement “pseudo-task" parallelism by annotating successive data-parallel for loops with nowait clause so that there is no barrier in between. This results in concurrent kernels running on the accelerator. In the context of stencil computation, such parallelism is not prevalent since the output of one process is typically input to the next, thus requires a global synchronization in between.

3.4 Mint Program Example

The Mint program for solving the 7-point heat equation, presented in Eqn 2.1 appears in Table 3.1. Note that this program is also legal C since a standard compiler will simply ignore the directives. The programming effort required to annotate the code is modest. We only introduced 5 pragmas and no existing source code was modified. The principal effort for the programmer is to (1) identify time consuming-loop nests that can benefit from acceleration and (2) move data between host and accelerator. The Mint `copy` directives at Line 1 and 2 transfer the input arrays, U and $Unew$ to the accelerator memory. The Mint `parallel` at Line 3 starts the accelerated region and its scope ends at Line 16. Any data needed upon region exit should be transferred back to the host otherwise their content will be lost. Line 17 retrieves U back to the host.

The Mint `for` directive appearing at Line 7 enables the translator to parallelize the `for` loop nest on lines (8-16). The `nest(all)` clause specifies that all loops should be parallelized. The `tile` clause specifies how the iteration space of a loop nest is to be subdivided into tiles. A tile will be assigned to a group of thread. Based on the `chunksize` clause, each thread in the

Table 3.1: Mint program for the 7-point heat solver

```
1  #pragma mint copy(U,toDevice,(n+2),(m+2),(k+2))
2  #pragma mint copy(Unew,toDevice,(n+2),(m+2),(k+2))
3  #pragma mint parallel
4  {
5    int t=0;
6    while( t++ < T ){
7  #pragma mint for nest(all) tile(16,16,64) chunksize(1,1,64)
8     for (int z=1; z<= k; z++)
9      for (int y=1; y<= m; y++)
10      for (int x=1; x<= n; x++)
11       Unew[z][y][x] = c0 * U[z][y][x] + c1 * (U[z][y][x-1] + U[z][y][x+1] +
12                        U[z][y-1][x] + U[z][y+1][x] + U[z-1][y][x] + U[z+1][y][x]);
13    double*** tmp;
14    tmp = U; U = Unew; Unew = tmp;
15   }//end of while
16 }//end of parallel region
17 #pragma mint copy(U,fromDevice,(n+2),(m+2),(k+2))
```

thread group gets its work assignment. In this example, the compiler will create tiles with the size of $16 \times 16 \times 64$ and each accelerator thread will be responsible for computing 64 iterations in the z-dimension of a tile. In Chapter 4, Section 4.2.6 will present the generated host code and Section 4.2.7 will present the generated accelerator code for the Mint program solving the 7-point heat equation.

3.5 Performance Programming with Mint

The Mint programmer can optimize code for acceleration incrementally with modest programming effort. The code optimization is accomplished by supplementing a Mint `for` pragma with clauses and by specifying various compiler options. We have discussed the for-loop clauses in Section 3.3.2. Next we discuss the compiler options that can potentially improve the performance. We leave the discussion about how these optimizations are implemented in the compiler to Chapter 5.

3.5.1 Compiler Options

As stated previously in system assumptions, an accelerator has a fast on-chip memory in addition to slow off-chip memory. Mint provides a few compiler options listed in 3.2 to

Table 3.2: Summary of Mint Compiler Options

Compiler Option	Description
-register	utilizes register file to reduce global memory
-shared[1-8]	utilizes shared memory to optimize references to nearest neighbors [#] sets an upper limit on the amount of shared memory allowable for a kernel
-preferL1	favors L1 cache over shared memory (only applicable to certain accelerators)

manage on-chip memory at a high level, often in conjunction with the `for`-loop clauses. The *register* option enables the Mint register optimizer, which takes advantage of the large register file residing on the on-chip memory on the accelerator. The optimizer places frequently accessed array references into registers. Since the content of a register is visible to one thread only, this option improves accesses to the central point of a stencil, but not the neighboring points that are shared by other threads. For that purpose, Mint offers the shared memory optimizer triggered by the *shared* flag. The shared memory optimizer detects the sharable references among threads and chooses the most frequently accessed array(s) to place in shared memory. This optimization is shown in [Mic09, DMV+08] to be particularly beneficial for the stencil computation because of the high degree of sharing among threads.

In applications that have many arrays, deciding *which* arrays to map to shared memory–even *how many*–is tricky. The Mint optimizer automatically chooses some arrays to assign to shared memor, by using an algorithm to rank arrays referenced in a kernel based on their potential reduction in memory accesses when placed in shared memory. Mint also takes into account the amount of shared memory needed by an array in its ranking algorithm. It picks the arrays that maximize the total reduction in memory references and minimize the shared memory usage. Mint processes 3D input grids as 2D planes because the limited size of shared memory. To indicate up to *how many* planes can reside in shared memory, the programmer can set a value from 1 to 8 to the shared flag (e.g *shared=1*, default is 8). The compiler takes into account the limit set by the user when selecting arrays for shared memory. The details of the selection algorithm is explained in Chapter 5.

Certain accelerators (such as Fermi-based GPUs) come with a configurable on-chip memory as software- or hardware-managed. Mint allows the programmer to favor hardware-managed memory over software-managed memory with a compiler option, namely *preferL1*. This option is different from the Mint *shared* option, as it determines the amount of on-chip

Table 3.3: Summary of Mint Directives

Format and Optional Clauses	Description
`#pragma mint parallel` `{ }`	indicates the scope of an accelerated region
`#pragma mint for \`	marks data parallel for-loops
`nest(# | all)`	depth of loop parallelism
`tile(`t_x`, `t_y`, `t_z`)`	partitioning of iteration space
`chunksize(`c_x`, `c_y`, `c_z`)`	workload of a thread in the tile
`reduction(operator:var_name)`	reduction operation on the specified variable.
`nowait`	enables host to resume execution
`#pragma mint copy(src | dst,`	transfers data between host and accelerator
`toDevice | fromDevice,`	
N_x`, `N_y`, `N_z`, ... )`	
`#pragma mint barrier`	synchronizes host and accelerator
`#pragma mint single` `{ }`	serial execution on the accelerator

memory which is set aside as shared memory. The rest will be L1 cache. The total amount of shared memory needed by the application depends on the tile size, the number of variables kept in shared memory. If the program demands more shared memory than available, Mint will issue an appropriate message and terminate.

3.6 Summary

With the non-expert user in mind, simplicity is our principal design goal for the Mint programming model. A pragma-based programmer interface is a natural way of meeting our requirement. Mint is minimalist and employs just five directives that are sufficient to accelerate many applications. Table B.2 summarizes the Mint directives. The principal effort for a Mint programmer is to identify time consuming-loop nests that can benefit from acceleration, and data motion between host and accelerator.

The Mint model enables the programmer to write applications capable of using accelerators, without the need to learn how the underlying architecture works. Mint helps the user manage the separate host and device memory spaces. The programmer specifies transfers at a

high level through the Mint `copy` directive, avoiding storage management and setup. Mint handles preparation of kernel parameters, passing them to the accelerator, and offloading the kernel to the accelerator.

Moreover, the Mint programmer does not need to explicitly manage accelerator threads nor the accompanying locality issues. These are handled by the Mint translator, with some guidance from directive clauses and compiler options. The workload decomposition clauses hide significant performance programming, enabling non-experts to incrementally tune the code for data locality without entailing disruptive reprogramming. For example, they can manage the mapping of loop iterations to thread indices and work assignment to threads. In addition, Mint provides a few compiler options that help reducing in memory references in favor on-chip memory. We will demonstrate how a programmer can tune an application via directives and compiler options in the result chapter (Chapter 7). Appendix A describes how the Mint distribution can be obtained, and Appendices B and C provide a cheat sheet and a performance tuning guide for a novice programmer.

Acknowledgements

This chapter, in part, is a reprint of the material as it appears in International Conference on Supercomputing 2011 with the title “Mint: Realizing CUDA performance in 3D Stencil Methods with Annotated C" by Didem Unat, Xing Cai and Scott B. Baden. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Mint Source-to-Source Translator

The Mint translator has two main stages. The first stage, *Baseline Translator*, transforms C source code with Mint annotations to unoptimized CUDA, generating both device kernel code and host code. The second stage performs architecture- and domain-specific optimizations. The output of the translator is a CUDA source file, which can be subsequently compiled by `nvcc`, the CUDA C compiler. The Mint work flow is illustrated below. This chapter covers only the baseline translator.

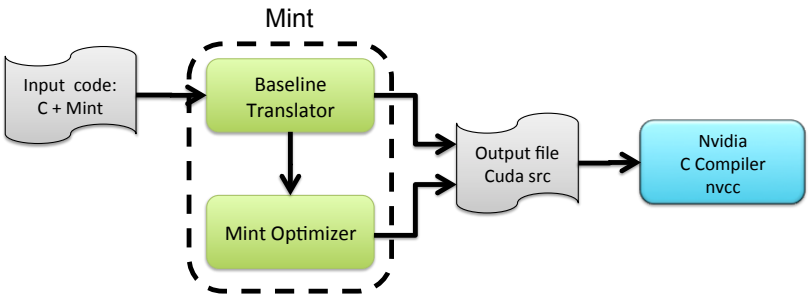


Figure 4.1: Mint Translator has two main stages: Baseline Translator and Optimizer. Mint generates CUDA source file, which can be subsequently compiled by the Nvidia C compiler.

4.1 ROSE Compiler Framework

To construct our source-to-source translation and analysis tools, we used the ROSE compiler framework[QMPS02, ros]. ROSE is an open source software developed and maintained at Lawrence Livermore National Laboratory. The framework accepts multiple source languages

including C, C++, Fortran, Python and Haskell and supports UPC, OpenMP and MPI. ROSE reads the input source code and constructs an intermediate representation (IR) called Sage III. IR is a low level form of the source code, which is independent of the input language and target architecture. All the analysis and transformations are performed on the Abstract Syntax Tree (AST), which is converted from the IR after front-end parsing. There are around 700 different AST nodes in ROSE: nearly half of them are dedicated to source code representations and the other half to binary analysis. ROSE supports transformation and analysis tools including loop unrolling, loop normalization, def-use analysis, constant propagation and many more. The ROSE backend unparses the modified AST to a source file by keeping original comments and control structure. The output code can be then compiled by a backend compiler (e.g gcc, icc). ROSE is released under a BSD license and has been tested under Linux and Mac OS X.

Currently, the framework does not fully support the CUDA API though the remaining support is on the way. Some of the supported CUDA features include the kernel launch syntax (< < <..> > >) and the `__global__` and `__shared__` keywords. On the other hand, we had to treat some CUDA-specific preserved variables (e.g. threadIdx.x) as strings. This treatment makes it hard to query such variables on the AST when performing optimizations on the generated code. Unfortunately the incompleteness in ROSE's support cost us considerable development time and the lack of full CUDA support significantly increased the number of lines of code in the translator. However, the ROSE team is aware of the issue and has plans to address it.

One of the most useful features supported by ROSE is outlining [LQVP09]. We utilized the *Outlining* tool to convert Mint for-loops into CUDA kernels. However, our outlining process required modifications to the ROSE's outliner. The original outliner outlines a basic block as a function by moving the basic block to the newly created function. In CUDA, the outlined function is a kernel and entails additional processing. Mint's Outliner is discussed in detail in Section 4.2.2. Moreover, some of the program analysis tools in ROSE didn't accommodate our needs. We had to add more functionality such as def-use analysis for array expressions and complex index expressions in constant folding. Another place where we needed to tweak the ROSE compiler was in Mint's pragma handler. ROSE can process OpenMP pragmas but cannot parse Mint pragmas. Therefore, we wrote our own parser to process the directives and their arguments. We hope in the future ROSE will allow developers to easily define arbitrary directives and their rules. This feature will be extremely useful for developing other directive-based models within ROSE. In fact, the ROSE team has decided to refactor and merge some of the Mint code transformations back into ROSE so that more users can benefit from them.

There are other open-source alternatives to ROSE such as LLVM [LA04] and Cetus [iLJE03]. We picked ROSE because it allows us to work on the source file at a high level AST, closer to the programming language. It does not lose any information about the structure of the original source code during the transformation. LLVM targets generation of object code which makes it difficult to develop source-to-source translators. For example, it does not have the notion of loops. On the other hand, LLVM can be preferably used to develop low-level compiler optimizations. As of CUDA 4.1, Nvidia switched over to LLVM inside their C/C++ compilers for Fermi-based devices to generate PTX code, an assembly language used for Nvidia GPUs. Cetus is a java-based compiler framework used in compiler research including OpenMPC [LME09].

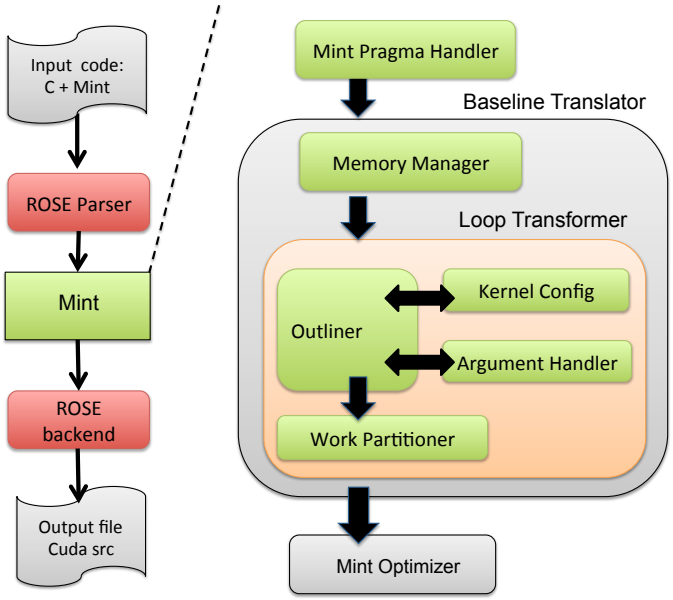


Figure 4.2: Modular design of Mint Translator and the translation work flow.

4.2 Mint Baseline Translator

Fig. 4.2 shows the modular design of the Mint translator and the translation work flow. The input to the compiler is C source code annotated with Mint pragmas. Once the translator has constructed the abstract syntax tree (AST), the *Pragma Handler* parses the Mint directives and clauses, and then it verifies that they are syntactically correct. Next, the translator queries the parallel regions containing data parallel for-loops. Directives in a candidate parallel region

```
1  nodeList = querySubTree(parallel_region,  V_SgPragmaDeclaration)
2
3  foreach node in nodeList
4    if(is Transfer Pragma(node))
5      params = get Transfer Parameters(node)
6      if(params.trfType == is Transfer to Device)
7           if(hostToDev.find(src_sym)) //found
8              get device name and other transfer parameters
9           else//not found
10             declare a device pointer outside  parallel region
11             allocate space on device memory
12             hostToDev[src_sym] = dev_sym
13          end if
14          generate Cuda Memory Copy Call
15     end if
16     else if (params.trfType == is Transfer from Device)
17         …
19   end if
19 end foreach
```

Figure 4.3: Pseudo-code showing how Mint processes copy directives inside a parallel region

go through several transformation steps inside the Baseline Translator: *Memory Manager*, *Outliner*, *Argument Handler*, *Kernel Config*, and *Work Partitioner*, which we discuss in turn in the following subsections.

4.2.1 Memory Manager

Vector arguments to kernels that are referenced in a parallel region need to be transferred to device memory. The *Memory Manager* handles the data transfers between the host and device. For each parallel region detected in the input code, the *Memory Manager* is called to process the copy pragmas. The manager first handles the pragmas preceding a parallel region. These pragmas express the data transfers from host to device. Secondly, it processes the data transfer requests inside a parallel region. These transfers can be from or to device. Then, it handles the copy pragmas proceeding the parallel region, which are the transfers from the device to host. Finally it frees all the allocated space for each variable on the device memory. To keep track of the mappings between the host variables and their device counterparts, the *Memory Manager* creates a table called *hostToDev* for each parallel region. This table is passed to subsequent translation stages.

A copy directive implicitly leads to a series of operations depending on where it occurs. An occurrence of a copy directive may result in (1) a declaration of a device variable, (2) allocation of its space, (3) deallocation of its space on the device, (4) a data transfer from host to

device, or (5) from device to host. Outside a parallel region a copy *toDevice* results in a declaration of a device variable, allocation of the space, and a transfer from host to device. The compiler generates a number for the device variable names with a *dev_* prefix and attaches the host name as the suffix (e.g. *dev*_1_*U*). This convention makes it easy for a programmer to debug and analyze the generated code. If a transfer *toDevice* occurs inside a parallel region, depending on whether it is the first transfer of that variable for the parallel region or not, the *Memory Manager* either only transfers its content or handles the declaration and allocation as well. The translator frees up the device memory upon the exit of a parallel region and cleans up the *hostToDev* table. A copy *fromDevice* outside a parallel region results in a data transfer from the device to host first, and then deallocation of the space.

Fig.4.3 shows the pseudo-code for how Mint processes copy directives inside a parallel region. The *querySubTree()* is an interface in ROSE to traverse a subtree of the AST starting from a given root node. It collects all the variants of a given type. In the pseudo-code the root node is a *parallel_region* and the AST node variant is a *PragmaDeclaration*. The return value is a node list of all the Sage IR nodes that are pragma declarations. Since ROSE does not recognize the Mint directives, we cannot query a *MintCopyPragmaDeclaration* directly. Therefore, we check whether the pragma declaration is a Mint copy directive or not (Line 4), then we use our custom parser to parse the transfer parameters in Line 5. Line 7 uses the *hostToDev* table to query whether the variable has already a device counterpart. If the variable is found, we retrieve its device name and dimension parameters. If the variable is not found in the table, the translator handles the declaration and allocation of the variable outside the parallel region and adds this variable to the table. Finally, we generate CUDA memory copy call at Line 14.

4.2.2 Outliner

After managing data transfers, the translator searches all the parallel for-loops inside a parallel region. The *Outliner* outlines each candidate parallel for-loop into a function: a CUDA kernel. It has two main sub-components: *generateFunction* and *generateCall*. The generateFunction moves the body of the loop into a newly-created `__global__` function. The generateCall replaces the statement that outliner vacates including the original `for` directive with a kernel launch as illustrated in Fig.4.4.

The Outliner's generateFunction creates an argument list based on the variable list it gets from the *Argument Handler*. If a variable is a vector, then it gets the device name from the *hostToDev* table and changes the type to a CUDA data type. GenerateFunction then creates a

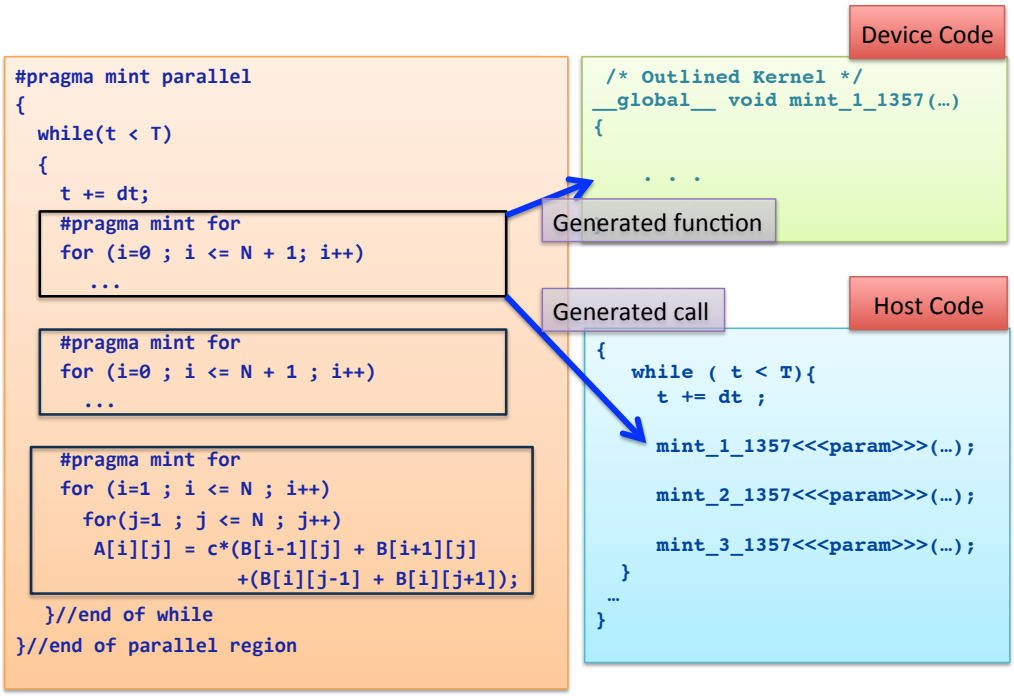


Figure 4.4: Outliner outlines each parallel for-loop into a CUDA kernel.

function skeleton with the function name, `void` return type[1], its parameters and scope, which is the global scope of the original loop body. We generate a string as the function name and add the *mint_* prefix to distinguish the Mint-generated kernels. The generateCall routine in Outliner generates the kernel launch code. A kernel launch has execution parameters specific to CUDA such as grid and block size. These parameters are declared and initialized in *Kernel Config*, which we discuss next.

4.2.3 Kernel Configuration

With the help of the `mint for` directive and attendant clauses, the *Kernel Config* stage determines the kernel configuration parameters, i.e. CUDA thread block and grid sizes. These parameters must be declared and initialized before the kernel launch and are passed to the Outliner. The number of blocks in each dimension is determined based on the iteration space of the loop and the value of the `nest` clause. The *Kernel Config* finds the upper and lower bounds of the loop and builds a range expression. Then, this range expression is used to build another expression to calculate the number of blocks in each dimension. For example:

[1]The return type of a CUDA kernel is always `void`.

```
numBlockX = ceil( range / tileDim.x )
```

Depending on the value of the `nest` clause, the loop range used in calculation differs.

- If `nest` = 1, the outermost loop range is used to compute number of blocks in X-dim. The number of blocks in other dimensions is set to 1.
- If `nest` = 2, the outermost loop range is used to compute number of blocks in Y-dim. The second outermost loop range is used to compute number of blocks in X-dim.
- if nest = 3, the outermost loop range is used to compute number of blocks in Z-dim. The second outermost loop range is used to compute number of blocks in Y-dim. The third outermost loop range is used to compute number of blocks in X-dim. Under CUDA, a grid of thread blocks can not have more than 2 dimensions. We emulate 3D grids by mapping two dimensions of the original iteration space onto one dimension of the kernel and create numBlockY * numBlockZ many blocks in Y-dim:

  (e.g `dim3 gridDim(numBlockX, numBlockY * numBlockZ)`).
- if nest $>$ 3, we currently do not parallelize the 4th or more dimension. The loops with nesting depth more than 3 should be annotated with nest=3. Either innermost or outermost three loops will be parallelized depending on where the programmer inserts the directives.

The following code fragment shows an example annotation for a triple nested loop and the generated code for its kernel configuration. The first 3 lines in the kernel configuration computes the number of blocks in each dimension by using the loop range expressions. For example, the loop range expression for the innermost loop is *upper - lower bound*, which is *n-1+1*. (Plus 1 is due to the equality sign in the upper bound). The range expression is divided by the tile size in the corresponding dimension to find the number of blocks. The necessary adjustment is made if the range is not divisible by the tile size. Finally, the *Kernel Config* declares the thread block size at Line 5 and then declares the grid size in the last line.

```
1 #pragma mint for nest(3) tile(16,16,1)
2   for (int z=1; z<= k; z++)
3    for (int y=1; y<= m; y++)
4     for (int x=1; x<= n; x++)
```

```
1 //kernel configuration
2     int num3blockDim_1_1527 = (k-1+1) \% 1 == 0?(k - 1 + 1) / 1 : (k - 1 + 1) / 1 + 1;
3     int num2blockDim_1_1527 = (m-1+1) \% 16 == 0?(m - 1 + 1) / 16 : (m - 1 + 1) / 16 + 1;
4     int num1blockDim_1_1527 = (n-1+1) \% 16 == 0?(n - 1 + 1) / 16 : (n - 1 + 1) / 16 + 1;
5     dim3 blockDim_1_1527(16,16,1);
6     dim3 gridDim_1_1527(num1blockDim_1_1527,num2blockDim_1_1527*num3blockDim_1_1527);
```

4.2.4 Argument Handler

The *Argument handler* works with the *Outliner*. It collects the list of all the variables referenced in the original loop body and determines which are local to the function and which need to be passed as arguments. The handler takes a basic block, which is a loop body, and returns a symbol list, which is the list of the variables to be passed. The ROSE compiler framework provides the interface to collect the list of variable symbols defined or referenced in a basic block. We classify symbols used in the basic block into two categories:

- *L* : locally declared variable symbols within loop body
- *U* : variable symbols referenced within loop body

  We take the set difference of the first two lists to determine which variables are to be passed.

- *P: U - L* : variable symbols referenced within loop body but not defined in the loop body

Note that the difference contains globally declared variables beyond the function's surrounding loop body. Normally an outlined function does not require global variables to be passed as arguments because such variables are visible to the outlined function (unless the function is put into a separate file). Under CUDA, we need to pass the globally declared variables in all cases because the device has a physically separate memory.

Naturally, all the parameters in the function argument become kernel call parameters. Depending on whether these parameters are vectors or scalars, they may require data transfers. The *Argument Handler* checks the *hostToDev* table the *Memory Manager* created to determine whether or not the programmer requested the transfer of vector variables via the Mint `copy` directives. If not, the compiler will issue an error message and terminate the translation.

4.2.5 Work Partitioner

The *Work Partitioner* inserts code into the generated kernel body to compute global thread IDs. It also rewrites references (i.e array subscripts) to original `for` loop indices to use

these global thread IDs instead. The loop iteration space is mapped one-to-one onto physical CUDA threads and the loop statement is replaced by an if-statement to check the array boundaries. When `chunksize` is set to 1, each thread is assigned to compute a single data point. Otherwise, the compiler inserts a serial loop into the kernel so that the thread can compute multiple points in the iteration space.

```
 1 loopList = querySubTree(loopNest,  V_SgForStatement)
 2 serialLoops = loopList.size() - nest
 3 parallelLoopNo = 1
 4 foreach cur_loop in loopList
 5   if (serialLoops > 0 )
 6     skip cur_loop
 7     serialLoops--
 8   else
 9     forLoopNormalization(cur_loop)
10     threadIndexVar = threadIndexCalculation ( cur_loop, clauseList, parallelLoopNo)
11     loopIndexVar = getLoopIndexVariable(cur_loop)
12     replaceVariableReferences(cur_loop, threadIndexVar, loopIndexVar )
13     replaceForLoopWithIfStatement(cur_loop)
14     parallelLoopNo++
15   end if
16 end foreach
```

Figure 4.5: Pseudo-code for the work partitioner in the translator.

Fig.4.5 shows very high level pseudo-code for the work paritioner. Line 1 starts by querying for-statements in the structured block proceeding a Mint-for directive. It keeps them in a loop list. A loop list contains a pointer to all the loops in a loop nest. Line 4 processes the list beginning from the innermost loop because an outermost loop is closer to the root node on the AST tree. Any modification to the outers affects the entire subtree, thus breaking the loop list iterator. The value of the `nest` clause provided by the programmer determines how many of the outer loops are parallelizable. Lines (5-7) skip the serial loops if the number of loops in the nested loop is greater than the number of parallelizable loops. Line 9 performs the ROSE's loop normalization on the current loop, which simplifies the data analysis. The ROSE's implementation of loop normalization moves the index declaration outside of the loop. It modifies the loop conditional to either $<=$ or $>=$ operation (e.g $i < N$ is normalized to $i <= (N-1)$). If possible, it replaces the increment/decrement expression to an increment operation (e.g $i- = s$ becomes $i+ = -s$). Lastly, it folds any constants appearing in the loop test and increment expressions.

At Line 10 we calculate the thread index expressions using the for-loop *clauseList* data structure. The clauseList contains the tile and chunksize information, both gathered from the

Table 4.1: Mint program for the 7-point 3D stencil

```
1  #pragma mint copy(U,toDevice,(n+2),(m+2),(k+2))
2  #pragma mint copy(Unew,toDevice,(n+2),(m+2),(k+2))
3  #pragma mint parallel
4  {
5    int t=0;
6    while( t++ < T ){
7  #pragma mint for nest(all) tile(16,16,64) chunksize(1,1,64)
8     for (int z=1; z<= k; z++)
9      for (int y=1; y<= m; y++)
10      for (int x=1; x<= n; x++)
11       Unew[z][y][x] = c0 * U[z][y][x] + c1 * (U[z][y][x-1] + U[z][y][x+1] +
12                        U[z][y-1][x] + U[z][y+1][x] + U[z-1][y][x] + U[z+1][y][x]);
13    double*** tmp;
14    tmp = U; U = Unew; Unew = tmp;
15   }//end of while
16 }//end of parallel region
17 #pragma mint copy(U,fromDevice,(n+2),(m+2),(k+2))
```

Mint annotations. The thread index variable is used to locate the thread work assignment in the iteration space. We next replace all the occurrences of the loop index variable with thread index variable (Line 12). Lastly, Line 13 removes the for-loop statement and generates an if-statement with a condition that checks whether the thread index variable is within the range of the loop start and end conditions. Naturally, the for-loop body becomes the body of the if-statement. In the if-body all the multidimensional array references must be flattened because in CUDA a multi-dimensional array should be represented as a single dimensional array. For example, an expression $A[i][j]$ becomes $A[i*N+j]$ under CUDA. We leave the array flattening to the very end because it is easier to do index analysis on multi-dimensional arrays when we perform the optimizations.

Next, we will go through generated host and device code examples via a simple Mint program, the 3D heat equation solver provided in Table 4.1.

4.2.6 Generated Host Code Example

Table 4.2 shows the host code generated by the Mint translator for the 7-point 3D stencil example. Lines (1-19) perform memory allocation and data transfer for the variable *U*, corresponding to line 1 in Table 4.1. the Mint compiler uses CUDA *pitched* pointer type and *cudaMalloc3D* to pad storage allocation on the device to ensure hardware alignment requirements are met [Nvi10a]. The extent field of the *cudaMemcpy3D* defines the dimensions of the transferred area in elements and it is set at the *cudaExtent* declaration. Lines (22-28) compute the kernel configuration parameters based on values provided by the user (i.e. via for-loop clauses), if there are any, else it chooses default values. Under CUDA, a grid of thread blocks can not have more than 2 dimensions. A common trick in CUDA is to emulate 3D grids (lines (26-28)) by mapping two dimensions of the original iteration space onto one dimension of the kernel. Line (31) launches the kernel and line (32) is a global barrier across all threads employed in the kernel launch. Lines (37-39) perform the pointer swap on the device pointers.

4.2.7 Generated Device Code Example

Table 4.3 shows the unoptimized kernel generated by Mint. All the memory accesses pass through global memory. Lines (5-7) unpack the CUDA pitched pointer *U*, while lines (12-22) compute local and global indices using thread and block IDs. Lines (24-26) are `if` statements derived from the `for` statements in the original annotated source (lines 9-11 in Table 4.1). Finally, lines (27-29) perform the stencil mesh sweep on the flattened arrays. In CUDA, multi-dimensional indexing works correctly only if the `nvcc` compiler knows the pitch of the array at compile time. Therefore, the translator converts such indices appearing in the annotated code into their 1D equivalents.

4.2.8 Chunking

In the generated kernel code shown in Table 4.3, each CUDA thread updates a single element of `Unew`. However, there is a performance benefit to aggregating array elements so that each CUDA thread computes more than one point. Mint allows the programmer to easily manage the mapping of work to threads using the `chunksize` clause. For example, the following annotation would give 32 elements to a thread:

```
#pragma mint for nest(3) tile(16,16, 32) chunksize(1,1,32)
```

The code fragment below shows part of the generated kernel when the programmer sets a chunking factor in the z dimension, the 3^{rd} argument of the `chunksize` clause. As an optimiza-

Table 4.2: Host code generated by the Mint translator for the 7-point 3D stencil input.

```
1  /* Mint: Replaced Pragma: #pragma mint copy( U, toDevice,( n + 2 ),( m + 2 ),( k + 2 ) ) */
2    cudaError_t stat_dev_1_U;
3    cudaExtent ext_dev_1_U = make_cudaExtent(((n+2)) * sizeof(double ),((m+2)),((k+2)));
4
5  /* Mint: Malloc on the device */
6    cudaPitchedPtr dev_1_U;
7    stat_dev_1_U = cudaMalloc3D(&dev_1_U,ext_dev_1_U);
8    if (stat_dev_1_U != cudaSuccess)
9      fprintf(stderr,"%s\n",cudaGetErrorString(stat_dev_1_U))U;
10
11 /* Mint: Copy host to device */
12   cudaMemcpy3DParms param_1_dev_1_U = {0};
13   param_1_dev_1_U.srcPtr = make_cudaPitchedPtr(((void *)U[0][0]),((n+2)) * sizeof(double )
         ,((n+2)),((m+2)));
14   param_1_dev_1_U.dstPtr = dev_1_U;
15   param_1_dev_1_U.extent = ext_dev_1_U;
16   param_1_dev_1_U.kind = cudaMemcpyHostToDevice;
17   stat_dev_1_U = cudaMemcpy3D(&param_1_dev_1_U);
18   if (stat_dev_1_U != cudaSuccess)
19     fprintf(stderr,"%s\n",cudaGetErrorString(stat_dev_1_U));
20  ...
21  while(t++ < T){
22  //kernel configurations
23       int num3blockDim_1_1527 = (k - 1 + 1) % 1 == 0?(k - 1 + 1) / 1 : (k - 1 + 1) / 1 + 1;
24       int num2blockDim_1_1527 = (m - 1 + 1) % 16 == 0?(m - 1 + 1) / 16 : (m - 1 + 1)/16 + 1;
25       int num1blockDim_1_1527 = (n - 1 + 1) % 16 == 0?(n - 1 + 1) / 16 : (n - 1 + 1)/16 + 1;
26       float invYnumblockDim_1_1527 = 1.00000F / num2blockDim_1_1527;
27       dim3 blockDim_1_1527(16,16,1);
28       dim3 gridDim_1_1527(num1blockDim_1_1527,num2blockDim_1_1527*num3blockDim_1_1527);
29
30 //kernel launch
31       mint_1_1527<<<gridDim_1_1527,blockDim_1_1527>>>(n,m,k,c0,c1,dev_2_Unew,dev_1_U,
             num2blockDim_1_1527,invYnumblockDim_1_1527);
32       cudaThreadSynchronize();
33       cudaError_t err_mint_1_1527 = cudaGetLastError();
34       if (err_mint_1_1527) {
35         fprintf(stderr,"In %s, %s\n","mint_1_1527",cudaGetErrorString(err_mint_1_1527));
36
37   double* tmp = (double*)ptr_dU.ptr;
38   ptr_dU.ptr = ptr_dUnew.ptr;
39   ptr_dUnew.ptr = (void*)tmp;
40
41  }//end of while
42  ...
```

Table 4.3: Unoptimized kernel generated by Mint for the 7-point 3D stencil input.

```
__global__ void mint_1_1527(int n,int m,int k,double c0,double c1,
                            cudaPitchedPtr dev_2_Unew, cudaPitchedPtr dev_1_U,
                            int num2blockDim_1_1527,float invYnumblockDim_1_1527)
{
  double* U = (double *)(ptr_dU.ptr);
  int _widthU = ptr_dU.pitch / sizeof(double );
  int _sliceU = ptr_dU.ysize * _widthU;
  ...
  float blocksInY = num2blockDim_1_1527;
  float invBlocksInY = invYnumblockDim_1_1527;

  int _idx = threadIdx.x + 1;
  int _gidx = _idx + blockDim.x * blockIdx.x;
  int _idy = threadIdx.y + 1;
  int _idz = threadIdx.z + 1;
  int blockIdxz = blockIdx.y * invBlocksInY;
  int blockIdxy = blockIdx.y - blockIdxz * blocksInY;
  int _gidy = _idy + blockIdxy * blockDim.y;
  int _gidz = _idz + blockIdxz * 16;

  int _indexU = _gidx + _gidy * _widthU + _gidz * _sliceU;
  int _indexUnew = _gidx + _gidy * _widthUnew + _gidz * _sliceUnew;

  if (_gidz >= 1 && _gidz <= k)
    if (_gidy >= 1 && _gidy <= m)
      if (_gidx >= 1 && _gidx <= n){
          Unew[_indexUnew] = ((c0 * U[_indexU]) + (c1 * (((((U[_indexU - 1] + U[_indexU +
               1])
                           + U[_indexU - _widthU]) + U[_indexU + _widthU])
                           + U[_indexU - _sliceU]) + U[_indexU + _sliceU])));
        }
}
```

tion, the translator moves `if` statements outside the `for` statement. It also computes the bounds of the `for`-loop.

```
1 if (_gidy >= 1 && _gidy <= m)
2   if (_gidx >= 1 && _gidx <= n)
3     for (_gidz = _gidz; _gidz <= _upper_gidz; _gidz++)
4       Unew[indUnew] = c0 * U[indU] + ...
```

Chunking affects the kernel configuration (i.e. size of the thread blocks) by rendering a smaller number of thread blocks with “fatter" threads. This clause is particularly helpful when combined with on-chip memory optimizations because it enables re-use of data. The reason will be explained in more detail in Section 5.8 of Chapter 5.

4.2.9 Miscellaneous

CUDA limits the total size of the `__global__` function parameters to 256 bytes. In most cases this limit does not pose any restriction. However, it becomes an issue for an application using several arrays and scalars. Mint automatically takes care of such problems. The translator uses the *cudaPitchesPtr* data structure for vector variables. Each *cudaPitchesPtr* has a size of 32 bytes which means that only 8 such arguments can be passed to a kernel. Otherwise the generated code would trigger a compile time error because of the argument limit. In such cases, the Mint translator overcomes this restriction by packing arguments into a C-struct. The translator takes the device addresses returned from *cudaMalloc()* and writes them into a host-side C structure which is then copied to the device. In turn, the device unpacks the struct to obtain the device addresses. To avoid several struct allocations on the device the translator defines one struct per parallel region, and includes all the device pointers used in that parallel region in the struct. On the kernel code side however only the vectors used in that kernel are unpacked. Note that the programmer is not aware whether Mint uses a struct or not. The programmer does not need to modify the input source. This feature is enabled when the argument size exceeds the 256 byte limit.

Table 4.4 illustrates the Mint-generated code for the host-side C structure and its usage. Line 3 creates a host-side C-struct. Line 11 defines a struct variable on the host. Lines 14-16 store the device addresses to the struct fields, which follow by allocation and memory transfer of the struct. The code launches two kernels that take the struct as an argument. As shown in Table 4.5, the kernel `mint_3_1527` unpacks all three struct fields into the corresponding vectors.

Table 4.4: Mint-generated code for the host-side C struct to overcome the 256 byte limit for CUDA function arguments.

```
1  //Host Code:
2  //create a host-side C-struct
3  struct mint_1_1527_data
4  {
5    cudaPitchedPtr dev_1_u1;
6    cudaPitchedPtr dev_2_v1;
7    cudaPitchedPtr dev_3_w1;
8  }
9
10 //define a struct variable on the host
11 struct mint_1_1527_data __out_argv1__1527__;
12
13 //write device addresses to the struct
14 __out_argv1__1527__.dev_3_w1 = dev_3_w1;
15 __out_argv1__1527__.dev_2_v1 = dev_2_v1;
16 __out_argv1__1527__.dev_1_u1 = dev_1_u1;
17
18 //define a struct variable on the device
19 struct mint_1_1527_data *dev__out_argv1__1527__;
20
21 //allocate the struct on the device memory
22 cudaMalloc(((void **)(&dev__out_argv1__1527__)),1 * sizeof(struct mint_1_1527_data ));
23
24 //copy the content from host to device memory
25 cudaMemcpy(dev__out_argv1__1527__,&__out_argv1__1527__,1 * sizeof(struct mint_1_1527_data )
       ,cudaMemcpyHostToDevice);
26
27 //pass the device struct as an argument to the kernel
28 mint_3_1527<<<gridDim_3_1527,blockDim_3_1527>>>(nxt,nyt,nzt,dev__out_argv1__1527__);
29 ...
30 //pass the device struct as an argument to the kernel
31 mint_2_1527<<<gridDim_2_1527,blockDim_2_1527>>>(nxt,nyt,nzt,dev__out_argv1__1527__);
32 ...
```

Table 4.5: Mint-generated code for the device-side that unpacks the C struct. The translator unpacks *u1, v1* and *w1* on the first kernel but only unpacks *u1* in the second because other vectors are not referenced in the second kernel.

```
1  //Device Code:
2  __global__ mint_3_1527(...)
3  {
4    //unpack the struct to obtain device addresses
5    cudaPitchedPtr dev_1_u1 = dev__out_argv1__1527__ -> dev_1_u1;
6    cudaPitchedPtr dev_2_v1 = dev__out_argv1__1527__ -> dev_2_v1;
7    cudaPitchedPtr dev_3_w1 = dev__out_argv1__1527__ -> dev_3_w1;
8    ...
9  }
10 __global__ mint_2_1527(...)
11 {
12   //unpack the struct to obtain device addresses
13   cudaPitchedPtr dev_1_u1 = dev__out_argv1__1527__ -> dev_1_u1;
14   ...
15 }
```

On the other hand, the second kernel only unpacks one of the variables because other vectors are not references in the kernel body.

4.3 Summary

This chapter described the details of the Mint baseline translator, which is built on the ROSE compiler framework. The baseline translator transforms C source code with Mint annotations to unoptimized CUDA code. The input code goes through several transformation inside the baseline translator. The copy directives are handled by the *Memory Manger*, which performs the data transfers between the host and device memory. The *Outliner* step outlines a candidate parallel for-loop into a CUDA kernel. The *Argument Handler* assists to the *Outliner* by determining the argument list that should be passed to the newly generated CUDA kernel. The next step is the *Kernel Config* where the translator configures the thread block and grid sizes, which are derived from the tile and chunksize clauses provided by the user. Lastly, the *Work Partitioner* maps the loop indices to thread indices and arranges the work division between threads.

The output code of the *Baseline Translator* makes all memory references through device memory. If one of the optimization flags is turned on, the *Mint optimizer* performs stencil method-specific optimizations on the generated code. Such optimizations are discussed in the next chapter.

Acknowledgements

This chapter, in part, is a reprint of the material as it appears in International Conference on Supercomputing 2011 with the title “Mint: Realizing CUDA performance in 3D Stencil Methods with Annotated C" by Didem Unat, Xing Cai and Scott B. Baden. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Mint Optimizer

The Mint optimizer incorporates a number of optimizations that we have found to be useful for stencil methods, especially in 3D. When Mint optimizations are enabled, the translated code takes advantage of on-chip memory. In order to optimize for on-chip memory, we implemented a *stencil analyzer* that analyzes stencil structure. Based on this analysis, the optimizer then utilizes shared memory, L1 cache (if available) and registers to improve data locality. In this chapter, we discuss both the analysis and the optimization steps in detail.

To better understand the compiler optimizations, we begin this chapter with an overview of the optimization strategies for stencil methods. We manually ported several types of stencil computations to GPUs and explored device capabilities before implementing the auto-optimizer. This was useful for developing common optimization strategies and integrating them into the translator. After discussing optimization strategies in Section 5.1, we proceed with the compiler optimizations in Section 5.2.

5.1 Hand-Optimization of Stencil Methods

In this section, we describe optimization strategies for stencil methods. Although we present the strategies in the context of GPUs, the strategies are applicable to traditional multi-core architectures as well. The optimization strategies mainly focus on utilizing memory bandwidth because stencil methods are generally bandwidth-limited. We found that properly managed on-chip memory can reduce the DRAM references significantly and hence greatly improve performance.

5.1.1 Stencil Pattern

Depending on the numerical method, a stencil computation couples a specific set of data points surrounding the point to be updated. We refer to this set as a stencil pattern. As an example, Fig. 5.1 shows the data points required to compute a single point for 7-point, 13-point and 19-point stencils. In the figure, the primary point of interest is shown in black. We refer to the black point as the *center* and the xy-plane as the center plane. Other points in the center plane are called *off-center* points. We refer to the points that lie in the planes above the center plane as the top points and in the planes below the center plane as the bottom points. In order to update the center point in a 7-point stencil, we use the 6 nearest neighbors. The 13-point stencil extends the 7-point stencil to include points further than the nearest neighbors, but along the Manhattan directions only. The 19-point stencil is more compact; it chooses the nearest neighbors at the edges and corners.

Consider a naive implementation of the stencil computation that makes all memory references through device memory. For example, in a 7-point stencil computation, each thread performs seven loads and one store in order to update a single data point. Since many points access the same values, there is a high amount of reuse. Considering the accessing time memory is long, it is worth optimizing the stencil codes to reduce the number of global memory loads and stores. We can eliminate redundant accesses by reusing the data already brought to the on-chip memory. In the next section, we explain how we can improve the reuse.

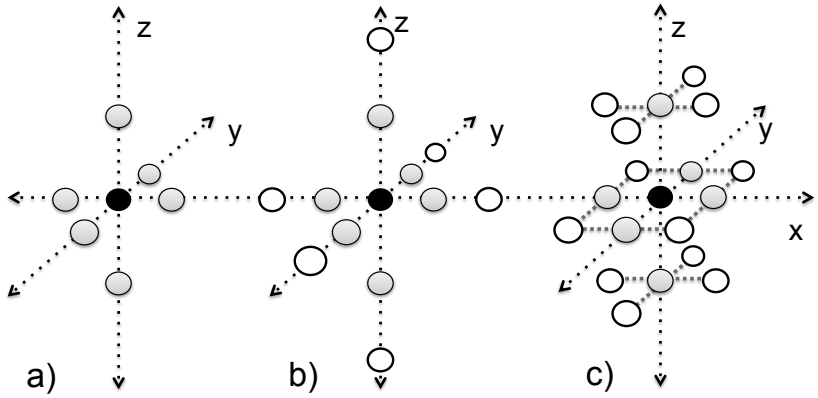


Figure 5.1: A stencil contains a specific set of data points in a surrounding neighborhood. The black point is the point of interest. a) 7-point stencil, b) 13-point stencil, c) 19-point stencil.

5.1.2 GPU Parallelization of Stencil Methods

A GPU device has two types of on-chip memory that can be explicitly managed at the software level: shared memory and registers. We can use both resources to buffer global memory accesses. However, we cannot fit the entire grid into on-chip memory because both shared memory and registers are small. The solution is to divide the grid into 3D tiles and process each tile as a series of 2D planes, as illustrated in Fig. 5.2. This way, a different CUDA thread will handle a different point in a 2D plane, and will be responsible for a unique data point in the mesh that it handles from global memory into on-chip memory.

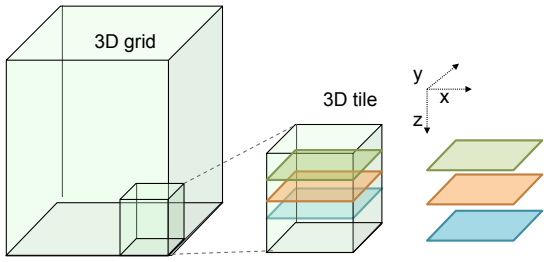


Figure 5.2: Divide 3D grid into 3D blocks and process each block plane by plane.

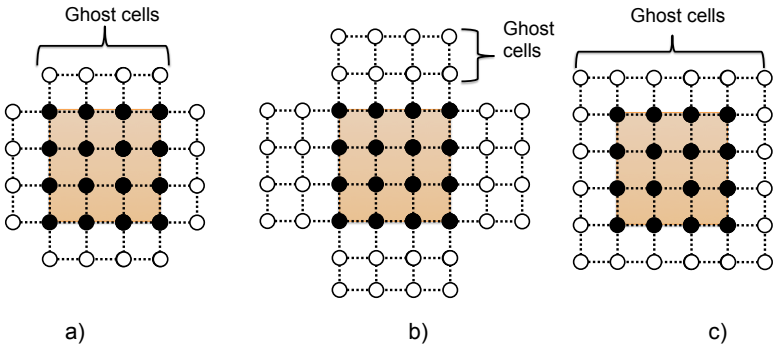


Figure 5.3: Ghost cells for a) 7-point stencil, b) 13-point stencil, c) 19-point stencil

Shared Memory. Threads that belong to the same thread block share the data brought into shared memory, but they do not have access to another block's data residing in shared memory. This is because shared memory is private to each thread block. For points located at the edges of a plane, some of the neighbors belong to another plane, which is processed by another thread block. These points are called *ghost cells*, as illustrated in Fig.5.3. The ghost values are usually kept in an additional set of grid points, around the plane so that the same

stencil operation is applied to every point in the interior plane.

A shared memory implementation needs to load ghost cells into shared memory. In order to handle ghost cells, we could create additional threads just for loading, but doing so would leave those threads idle during computation. In our experiments, this strategy resulted in poor performance. Instead, we let some of the threads handle the ghost cells in addition to computation. A thread block brings an xy-plane into shared memory with its respective ghost cells. Since the threads within a thread block share many data points on the xy-plane, the use of shared memory greatly reduces the number of global memory accesses. However, a thread needs a set of data points from a number of planes above and below the center. In addition to the center plane, we can bring all the required planes into shared memory as well. Although these planes are used by other thread blocks, bringing them into shared memory does not entirely reduce the redundant memory accesses because thread blocks do not share on-chip memory. In order to eliminate all redundant accesses, we employ a method called chunking, discussed next.

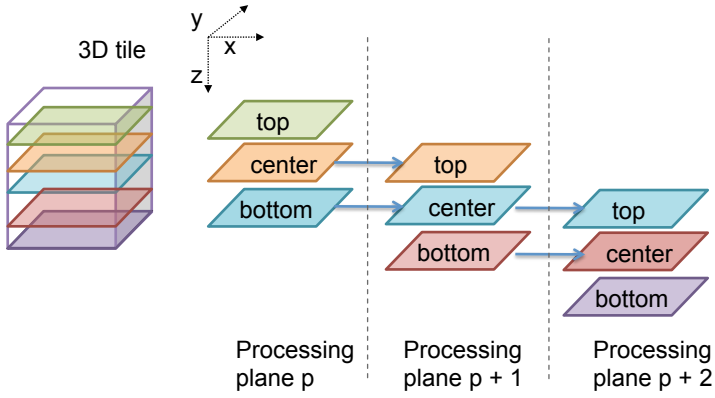


Figure 5.4: Chunking for a three plane implementation. A plane starts as the bottom, continues as the center and then as the top.

Chunking. We can reuse the data already in shared memory through software pipelining. We refer to this method as chunking, although it is also referred to as a sliding window [MK10, Mic09] or partial 3D blocking [RT00] optimization. Rivera and Tseng [RT00] proposed the method for traditional processors. Williams et. al [WSO+07] showed the effectiveness of the method on software-managed memory architectures such as the STI Cell Broadband Engine [KDH+05b]. Micikevicius [Mic09] applied the method to GPUs. We incorporated this optimization into our compiler through a simple clause, `chunksize`, saving a good deal of programming overhead.

The method iterates over the slowest varying dimension (z-dimension) by keeping a queue of planes and rotating them. Fig. 5.4 shows the chunking for a three plane implementation. A plane, which is loaded from global memory, starts as the bottom, then continues as the center and then as the top before it retires. This approach allows us to read data points only once and reuse them until they are no longer needed. Some stencils may require more than three planes in order to compute the center plane. Due to the limited size of shared memory, the best implementation may keep only some of the required planes in shared memory and read the rest from device memory or registers. Our experience with the stencil applications suggests that keeping three planes is typically sufficient because the stencil accesses are usually concentrated around the center point.

Registers. With the chunking optimization, we can eliminate all redundant memory accesses. To implement this optimization, we enlist the help of registers to alleviate pressure on shared memory. Registers can reduce the amount of shared memory required to support the chunking optimization. Another advantage of using registers is that an instruction executes faster if its operands are in registers [VD08]. This is because shared memory accesses require the operands to be placed in registers first, resulting in some overhead. Therefore, a thread should operate on registers as much as possible instead of operating on shared memory.

We can store the top, center and bottom points in registers. Since the registers are private to a thread and not visible to other threads, we still need a copy of the planes in shared memory to access off-center points, as shared memory is visible to all threads in the thread block. The use of registers can allow us to create more thread blocks and increase device occupancy. With the help of registers, we can reduce the amount of shared memory for certain kinds of stencils. For example, the 7-point or 13-point stencils (Fig 5.1) require that only one xy-plane of data be kept in shared memory at a time. The points accessed in the z-dimension can be in registers because there is no sharing along the z-dimension between threads.

When we utilize both registers and shared memory to implement the chunking optimization, a plane starts at the bottom in registers, continues at the center in shared memory and then to the top in registers again. This process is shown in Fig. 5.5. In the case of the 7-point stencil, while we are processing the plane p, we use three registers per thread to store the data values corresponding to the points in the z-dimension located in plane $p+1$, p and $p-1$. We iterate on p and shift the content of the registers. We load the new values from global memory in $p+1$ and retire $p-1$ at each iteration. A 13-point stencil would require 5 such registers to store data in $p \pm 2$ as well. By comparison, a 19-point stencil shares data in all three planes because of the

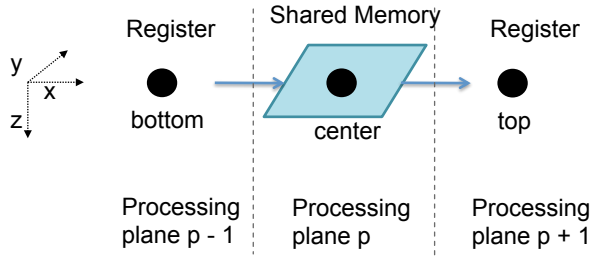


Figure 5.5: In chunking optimization, a plane starts as the bottom in registers, continues as the center in shared memory and then as the top in registers.

diagonal points, and thus requires all three planes in shared memory. Such a compact stencil will not be able to create more thread blocks with the help of registers. However, we can still benefit from registers to accelerate the instruction execution by storing the bottom, center and top points in registers.

Chunking in Other Dimensions: Since we let a thread compute multiple planes in the slowest varying dimension, we can assign more than one row in the next slowest varying dimension to a thread, reducing some of the index calculations. Allowing a thread to compute multiple elements, however, increases the number of registers used by the thread. Therefore, this optimization is beneficial only if there are enough available registers. It is disadvantageous to apply chunking to the fastest varying dimension due to the distruption in temporal locality across threads. Such locality is needed to ensure coalesced accesses to global memory in GPUs.

5.1.3 Common Subexpression Elimination

In a compact stencil such as 19-point or 27-point, some of the computed results are reused in more than one plane. Instead of recomputing those results, we keep the intermediate sums in registers. Unfortunately, a general purpose compiler fails to detect such a complex optimization because the optimization requires algorithmic transformation. The optimization reduces the number of flops performed per data point, and changes the order of the instructions. The opportunity does not exist in the stencils that refer to only one value from the top or bottom planes.

Fig. 5.6 visualizes the 19-point stencil on the left. On the right, we simplify the view of the stencil to show the edges that are reused in all three planes. When a plane is in the bottom, center, or top positions, the sum of the edges is needed. We keep the summed value in a register and therefore do not need to recompute it. This optimization reduces the flop count from 21 to 15

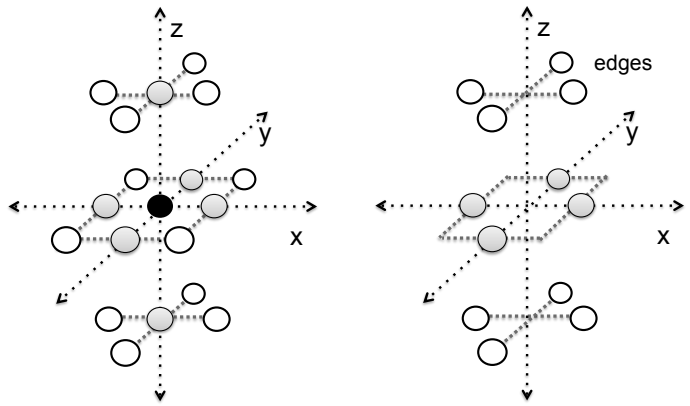


Figure 5.6: Visualizing 19-point stencil on the left and its edges on the right. We reuse the sum of the edges in top, center and bottom planes.

for the 19-point stencil. In this dissertation, we use the reduced flop counts in the calculation of Gflop/s rates for the 19-point kernels. More information about this optimization on a traditional multi-core architecture can be found in [DWV+09].

Next, we describe how we integrated the optimization strategies that we have just discussed into the Mint optimizer.

5.2 Overview of the Mint Optimizer

The baseline code generated by Mint naively makes all memory references through device memory, unless the optimization flags are enabled. When Mint optimizations are enabled, Mint tries to use on-chip memory (i.e shared memory, L1 cache, and registers) to reduce redundant accesses to global memory. After outlining an annotated nested-for loop into a CUDA kernel, the translator calls the optimizer to optimize the device code. The optimizer takes a kernel declaration and a clause list as arguments. The clause list contains information about the nest, tile and chunksize clauses provided by the user when they annotated the original loop.

Fig. 5.7 shows the basic steps that the optimizer follows for each kernel. First, the optimizer checks which optimization options have been selected. If the unroll flag is set, then the optimizer unrolls loops and applies the constant folding optimization, which we discuss in detail in Section 5.4. Next, the optimizer refers to the *Stencil Analyzer* to find access frequencies and candidate arrays for optimization, which we discuss in Section 5.3. Depending on which of the on-chip memory optimization options (*-preferL1, -register, and -shared*) are enabled by

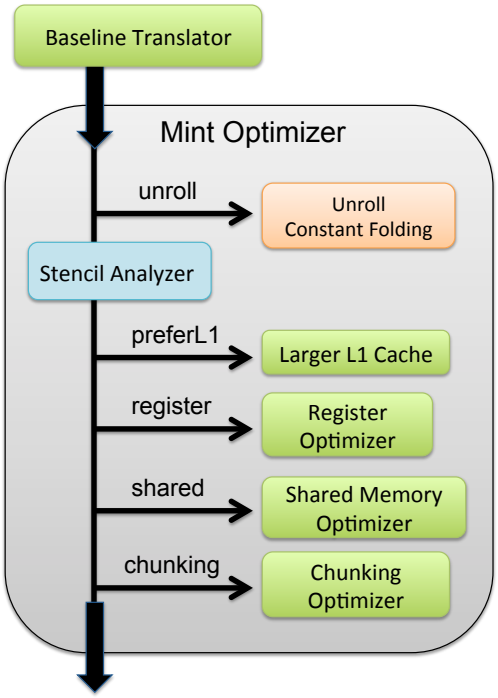


Figure 5.7: Workflow of Mint Optimizer

the user, the translator applies the optimizations in order. Chunking is not a compiler flag but is enabled by the `chunksize` clause specific to a Mint for-loop directive. This clause affects how we perform the shared memory and register optimizations on the kernel. We developed a separate subcomponent in the optimizer to handle discrepancies due to the chunksize clause, which subsequently applies the register and shared memory optimizations. We discuss each optimization in detail in Section 5.4 through 5.8. First we describe the *Stencil Analyzer*, because other components of the optimizer rely on it.

5.3 Stencil Analyzer

To optimize for on-chip memory re-use, the optimizer must analyze the structure of the stencil(s) appearing in the application. Based on this analysis, it then chooses an optimization strategy appropriate for the determined stencil pattern. The analyzer first collects all array reference expressions in the kernel and groups them by array name in order to rank the arrays based on their access frequency both for the register and shared memory optimizers. It also computers how many shared memory slots are available for optimization. We discuss these steps in turn.

5.3.1 Array Reference List

This component of the analyzer finds all array references in the kernel body and groups them according to array names. The function returns a map of array reference expressions where the hash key is the array name. For example,

- `E -> { E[i][j], E[i+1][j], E[i][j], ... }`
- `R -> { R[i-1][j] }`

Note that there might be multiple occurrences of the same reference in the list. We keep track of all instances of references since we are interested in the access frequencies. In the ROSE compiler framework, we can obtain the array references by querying SgPntrArrRefExp[1] for a given basic block. However, the query returns all array references including address references. For instance, it would return `E`, `E[i]` and `E[i][j]`. We skip the address expressions and store only the references to the array elements in the hash.

5.3.2 Shareable References

The array reference list contains all references made to arrays, without making any distinction whether they are nearest neighbor accesses or not. To define what is “near", we enforce a limit on the stencil pattern. The default maximum limit[2] is three on the xy-plane and one on the z-dimension on each side. In other words, Mint will consider references in the form of $[k \pm s][j \pm c][i \pm c]$ as near if $|s| <= 1$ and $|c| <= 3$. We will refer to these references as *shareable* because their values can be accessible from shared memory by other threads in the same thread block. The references that are not shareable will be accessed through global memory.

We further categorize the shareable references as center, off-center or up-down. Mint considers an index expression as center if it has zero offset in all dimensions ($[k][j][i]$). The off-centers are shareable references if they are in the form of $[k][j \pm c_j][i \pm c_i]$, where c_j or c_i are non-zero. The up-down references are again shareable references, which touch top or bottom planes ($[k \pm 1][j \pm c_j][i \pm c_i]$). Any reference that does not fall into one of these categories is considered to be a non-stencil access (hence not shareable) by Mint.

[1] Pointer Array Reference Expression
[2] Can be configurable.

5.3.3 Shared Memory Slots

If the shared memory optimizer is turned on, the Stencil Analyzer computes the number of shared memory *slots* the translator can use for optimization. A slot holds a 2D plane of a 3D input grid. The analyzer uses the tile size and data type (e.g, double, float) to compute the space that a plane will occupy. Then it divides the size[3] of shared memory by the size of the plane to compute the total number of slots that will fit. The user can set an upper limit on the number of slots allowable for an application. In that case, the Stencil Analyzer returns whichever is smaller:

```
# of slots = min(user's limit, (shared memory size / 2D plane size))
```

To indicate the maximum number of planes that can reside in shared memory, the programmer can set the value of the shared flag from 1 to 8 (e.g, shared=1, default is 8). For example, if the flag is set to 4, then Mint allows up to 4 planes to be placed in shared memory and assigns the corresponding slots to variables based on access frequencies. The planes may come from one or more arrays. For example, Mint could place 4 distinct arrays into shared memory or two distinct arrays, a single plane from one array, and three planes from the other. Next, we discuss how Mint grants these slots to the arrays.

5.3.4 Access Frequencies

This component of the analyzer takes an array reference list and the number of slots as an input, and returns two variable lists: a candidate list for shared memory and a candidate list for registers. Both lists are sorted in descending order of expected performance gain. We assume that the performance gain is proportional to the number of global memory reductions when the variable is placed in on-chip memory. We also take the amount of shared memory needed by an array into account in the sorting algorithm. The algorithm favors an array that requires less shared memory. The current implementation of the algorithm only grants up to three planes to a single variable because using more than three planes is uncommon.

For each array, the algorithm keeps a counter for each sharing category; center, off-center, up-down or non-stencil. For each reference, it increments the corresponding counter of an array based on the sharing category of the reference. A reference can belong to only one of the sharing categories. For simplicity, we keep a single counter for accesses to the top or bottom planes. This facilitates our decision about how many planes we allow for a variable. We either keep zero, one or three planes. There are cases where two planes might be sufficient. For example, there is no reference to the bottom, thus it is enough to spare one plane to the top and

[3]The size of shared memory is set to 16KB, but is configurable.

Table 5.1: Algorithm computing array access frequencies

```
1  foreach array in Array Reference List
2  {
3     keep a counter for center, off-center, updown and non-stencil
4     foreach reference in Reference List
5     {
6         find the sharing category based on the subscripts
7         update the counter
8     }
9     //there should be at least one off-center reference, otherwise we can just use registers
10    onePlaneList[array] = off-center != 0 ? (center + off-center) : 0;
11    centerList[array] = center;
12
13    //there should be at least more than 2 references
14    threePlanesList[array] = updown > 2 ? updown : 0;
15 }
16 //sort frequencies from highest to lowest
17 sortFreq(onePlaneList);
18 sortFreq(threePlanesList);
19 sortFreq(centerList);
```

one plane to the center. Our simplification is due to the fact that in most cases the computation will require points in the top and bottom planes together. We keep separate counters for the center and off-center accesses on the same plane because if the off-center counter is zero, then we can simply use registers rather than shared memory for the center point.

The algorithm employs three lists indexed by the array name: 1) *onePlaneList* contains the sum of the center and off-center counters for arrays. This list will be used to determine the variables that require only plane in shared memory, 2) *threePlanesList* contains the up-down counters for arrays. It will determine the variables that require three planes in shared memory, 3) *centerList* contains the center counters for arrays, which we will use for the register optimizer.

The pseudo-code in Table 5.1 shows the algorithm that computes the access frequencies and sorts them. The algorithm goes through each array in the array reference list and keeps a counter for each sharing category. Then, it looks at each reference in the reference list of the array and finds the sharing category of the reference based on its subscripts to update the corresponding counter. After visiting all references of an array, we add the center and off-center counters and store it in *onePlaneList*. In order for an array to have one plane in shared memory, it should have at least one off-center reference. Otherwise, we can simply use registers. We store the center counter in the *centerList* and up-down counter in the *threePlanesList*. To be eligible to have three planes in shared memory, we require that there are more than two points that lie on

the top and bottom planes, so it is worth allocating shared memory for that array.

After we count the sharing categories for each array, we independently sort the *onePlaneList*, *threePlanesList* and *centerList*. The frequencies in a list are sorted from highest to lowest. As a result, the algorithm sorts the arrays in descending order according to their potential benefit when the optimization is applied. For example, the first array in the *onePlaneList* after sorting is the most eligible candidate for shared memory optimization that will require a single plane. The *centerList* is directly sent to the register optimizer, which we discuss in Section 5.6. The shared memory optimizer, on the other hand, has to decide how many planes to grant to which variables. Next, we discuss variable selection algorithm in Mint.

5.3.5 Selecting Variables

This component of the analyzer takes the *onePlaneList, threePlanesList* and the number of shared memory slots as inputs and passes the result to the shared memory optimizer. The selection algorithm attempts to exhaust all available shared memory slots with candidate variables and returns a list of variables with their shared memory requirement. We will refer to this pair list as *selected*.

The pseudo-code of the selection algorithm appears in Table 5.2. The algorithm keeps two iterators; one for the *onePlaneList* and another for the *threePlanesList*. Assume that it_1 is the iterator for the *onePlaneList* and it_3 is the iterator for the *threePlanesList*. Let var_1 be the current variable that it_1 points to the onePlaneList and var_1 have a reference saving count ref_1. Let var_3 be the current variable that it_3 points to and var_3 have a reference saving count ref_3. The algorithm runs until it exhausts all the available slots, or the lists become empty (Line 5). For example, if assigning one plane to var_1 saves more references than assinging three planes to var_3, then we grant one slot to var_1, and advance the iterator it_1. If assigning three planes to var_3 saves more global memory references, then we grant three slots to var_3, and advance the iterator it_3. There are three cases for var_1 and var_3:

Case 1: If var_1 and var_3 are the same variables (Line 7), then we check whether or not assigning one plane is more advantageous than assigning three planes to that variable. In order to justify three planes, we compare ref_1 and ref_3. If ref_3 does not save twice the references that ref_1 saves (since it requires two more planes in addition to one plane), then we only assign one plane to that variable for now (this may change later). We decrement the slots by one and advance only the iterator of the onePlaneList. Otherwise, we decrement the slots by three, and advance both the iterators (Line 16-18).

Table 5.2: Variable selection algorithm for shared memory optimization

```
1  List selected;
2  it1 = onePlaneList.begin();
3  it3 = threePlanesList.begin();
4
5  while( slots > 0 && (!onePlaneList.end() || !threePlanesList.end())
6    {
7     case 1: var_1 and var_3 are the same variables
8        if(2 * ref_1 >= ref_3 ) //1-plane provides more savings
9         {
10          slots--
11          selected.insert(var_1, 1)
12          update it1, ref_1 and var_1
13        }
14       else //3-plane provides more savings
15        {
16          slots -= 3
17          selected.insert(var_3, 3)
18          update it1, it3, var_1, var_3, ref_1, and ref_3
19        }
20    case 2: var_3 is already in selected but granted with 1-plane
21       look ahead to next variable in the onePlaneList
22       if(ref_1 + ref_1_next > = ref_3) //two 1-planes provide more savings
23        { //same as Line 10-12
24        }
25       else //3-plane provides more savings
26        {
27          slots -= 2 //because we already gave one plane
28          selected.insert(var_3, 3)
29          update it3, var_3, and ref_3
30        }
31    case 3: default //var_1 and var_3 are different, and var_3 is not in selected
32       look ahead to next two variables in the onePlaneList
33       if(ref_1 + ref_1_next + ref_1_next2 >= ref_3) //three 1-plane provides more savings
34        { //same as Line 10-12
35        }
36       else //3-plane provides more savings
37        {
38           slots -= 3
39           selected.insert(var_3, 3)
40           update it3, var_3, and ref_3
41        }
42   } //end of while
```

Case 2: If the variable var_3 is already in the selected list because we previously encountered it in the onePlaneList and granted it one plane, we look ahead to the next variable in the onePlaneList to justify two more planes to var_3. Let's say var_1_next comes after var_1 in ranking. The combined savings by assigning one plane to var_1, and one plane to var_1_next should be less than the saving by assigning two more planes to var_3. Otherwise, we assign one plane to var_1.

Case 3: If the variables are not the same, and var_3 is not in the selected list, then we check if assigning one plane to three variables from the onePlaneList is more advantageous than assigning three planes to one variable from the threePlanesList. This time we look ahead to the next two variables; var_1_next and $var_1\_next^2$. If the savings from var_1, var_1_next and $var_1\_next^2$ when assigned one plane each is higher than the one from the var_3 when assigned three planes, then we reserve one slot for var_1. Otherwise we reserve three slots for var_3.

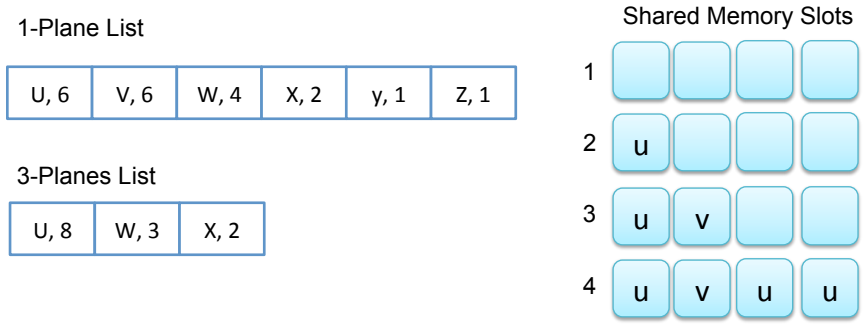


Figure 5.8: Filling shared memory slots with selected variables

Fig. 5.8 provides an example of the selection algorithm for shared memory. The onePlaneList and threePlanesList are created in the *Access Frequency* step of the Stencil Analyzer. Each entry in the list has the variable name and the number of references in the corresponding plane. Initially all the slots are empty. The selection algorithm takes one candidate from each list. Since the candidates are the same variable in this example, the algorithm falls into case 1. We assign one plane to u for now. Then, we compare the references of v from the onePlaneList and u from the threePlanesList. Since u is already in the selected list, the algorithm falls into case 2. We look ahead to the next variable, w, in the onePlaneList. v and w save 10 references in total. On the other hand, u's gain is 8. We grant one slot to v. We advance the iterator in the onePlaneList to w. In this case, we will assign the last two slots to the u because its saving (8 references) is greater than the combined savings of w and x (4 + 2). Note that if had only three shared memory slots available, then the algorithm would assign one plane each to u, v and w and save 16 references, because we did not have two slots for u.

5.3.6 Offset Analysis

Stencils differ in the required number of ghost cell loads depending on whether they are symmetric or cover a large neighborhood. The stencil pattern also affects the size of the shared memory reserved for the stencil. For example, instead of allocating a plane of size t_x by t_y to accommodate the ghost region, we need a plane of ($t_x + 2*$ offset) by ($t_y + 2*$ offset). The analyzer computes the ghost cell offset for a given stencil array. It considers only the shareable references in the xy-plane, where the pattern of array subscripts involves a central point and nearest neighbors only, that is, index expressions of the form $i \pm k$, where i is an index variable and k is a small constant. We conservatively choose the highest k in an asymmetric stencil for simplicity. If k is higher than the default value, then we set the offset to the default. If the offset of a reference is higher than the default value, we do not accommodate its ghost cells, because these accesses will go through global memory.

Next, we discuss the Mint compiler flags and how they are implemented in the Mint optimizer.

5.4 Unrolling Short Loops

The Stencil Analyzer assumes that array subscripts are direct offsets of the loop indices. This may not be always the case. The index expressions may be relative to another loop or expression, making it difficult for a compiler to analyze the stencil pattern. The code snippet in Table 5.3 shows an example, which sweeps a convolution window in the innermost two loops. An index to the *data* array is a function of l and k, which are functions of i and j (e.g., in Line 5). We implemented a preprocessing step to allow Mint to handle such cases, so that the stencil analyzer can effectively collect the stencil pattern appearing in the computation. Our preprocessing step is currently limited to unrolling serial loops that may appear in the nested loop. We unroll short vector loops which do not exhibit a lot of parallelism. The preprocessing step can be further improved to cover more complex index expressions.

If the *unroll* flag is on, the Mint translator examines the CUDA kernels generated by the baseline translator in order to make indices more explicit to the stencil analyzer. In order to eliminate the l and k indices from the loop body, Mint rewrites the references to arrays in terms of i and j. First, it determines the unrolling factor and recursively unrolls the loops. The unrolling factor is the difference between the upper and lower bounds of the loop. In this example, it is 5 (Lines 3-4). Mint completely unrolls such loops only if the relation between the indices is

Table 5.3: Index expressions to the *data* array are relative to inner loop indices.

```
1  for (i = 1; i < N ; i++){
2    for (j = 1; j < M ; j++){
3      for (k = i - 2; k <=i+2 ; k++){
4        for (l = j - 2; l <= j+2; l++){
5          dx = data[k][l+1] - data[k][l-1];
6          dy = data[k+1][l] - data[k-1][l];
7          ...
8        }}
9      out[i][j] = ...
10   }
11 }
```

based on a small constant. After unrolling, the translator replaces the instances of l and k with i and j respectively. This process introduces expressions such as $i \pm c_1 \pm c_2$, that involve the index variable and a number of constants. To simplify the index expressions, we apply constant folding. This optimization converts, if possible, the index expressions into the form $i \pm c$, where c is a small constant. After this step, the stencil analyzer can detect stencil patterns appearing in the loop body.

5.5 Cache Configuration with PreferL1

The flag *-preferL1* does not require any analysis or code transformation and is independent of other optimizations. Mint simply inserts `cudaFuncSetCacheConfig(kernel_name, cudaFuncCachePreferL1)` before the kernel launch. This suggests that the CUDA runtime to use the larger L1 cache configuration if possible, but the runtime is free to choose a different configuration if required. The flag has no effect on the 200-series, where the size of shared memory is fixed. Since the default configuration favors larger shared memory, we did not introduce a *preferShared* flag.

5.6 Register Optimizer

The Mint register optimizer takes advantage of the large register file residing on the device by placing frequently accessed array references into registers. Since the content of a register is visible to one thread only, register optimization enhances access to the central point of a stencil, but not the neighboring points shared by other threads. For that purpose, Mint uses

Table 5.4: Part of a kernel generated by the Mint translator after applying register optimization. The input code to the Mint translator is the Aliev-Panfilov model presented in Table 7.10.

```
1  float _rEprev = Eprev[_index2D];
2  float _rR = R[_index2D];
3  //written fist, no need to initialize
4  float _rE;
5
6  _rE = _rEprev + alpha * (Eprev[_index2D + _width] + Eprev[_index2D - _width]
7      - 4 * _rEprev + Eprev[_index2D + 1] + Eprev[_index2D - 1]);
8  _rE =_rE - (dt * (kk * _rE * (_rE - a) * (_rE - 1) + (_rE * _rR)));
9  _rR =_rR + dt * (epsilon + ((M1 * _rR) / (_rE + M2))) * (-_rR - (kk * _rE * (_rE - b-1));
10
11 //write back to global memory
12 E[_index2D] = _rE;
13 R[_index2D] = _rR;
```

the shared memory optimizer, which will be discussed shortly.

The register optimizer takes the candidate list from the stencil analyzer. The candidates are in descending order in terms of number of central references. Currently, we did not pose a limit on the number of arrays taking advantage of register optimization. As with shared memory optimization, a limit can be easy enforced via a compiler flag (e.g., -register=5), which will allow only a certain number of variables to reside in registers. Of course, in order to apply the register optimization, there should be at least one central reference. However, if both register and shared memory flags are turned on, even though there is no reference to the center, we still perform the register optimization because the central point may be referenced by a neighboring thread.

The optimizer first declares a scalar variable with the same type as the array element. Next, it reads the global memory reference into the scalar variable if the first reference is a read. If the variable is written before it is read, then there is no need to load the value from global memory. The optimizer replaces all the global memory references to the central point with the register references. If the variable is not read-only, then we write back the register content to global memory before the kernel exits.

The code snippet in Table 5.4 shows a part of a CUDA kernel generated by the Mint translator after applying register optimization. `E_prev` and `R` are read before being written. Therefore, we need to initialize their register counterparts (Line 1-2). Since `E` is written first, there is no need for initialization. Next, we replace the global memory references with the register references. Since `E` and `R` are modified, they require a write back to global memory

(Line 12-13). Overall, the kernel makes 21 references to the global memory; 6 for `Eprev`, 10 for `E`, 5 for `R`. As a result of register optimization, we perform just 5 global memory accesses for `Eprev`, only one for `E` and two for `R`. `Eprev` will benefit from the shared memory optimizer because of its stencil access pattern, which we discuss next.

5.7 Shared Memory Optimizer

The flag *-shared* enables the shared memory optimizer in the translator. Since the threads within a thread block reuse many data points, shared memory greatly reduces the number of global memory references. This optimizer works closely with the stencil analyzer, which provides the optimizer with a list of selected variables. The list contains array names and how many each planes each array needs. The optimizer goes through the selected variables and performs shared memory optimization on each. For each variable assigned to shared memory, the optimizer carries out a number of steps, which we describe next.

5.7.1 Declaration and Initialization of a Shared Memory Block

First we statically declare a shared memory block. Here are two examples using 1-plane or 3-planes:

```
__device__ __shared__ float _sh_U[TILE_Y+2][TILE_X+2];
__device__ __shared__ float _sh_U[3][TILE_Y+2][TILE_X+2];
```

The `__shared__` keyword indicates that the block should reside in shared memory[4]. This is followed by the element type, name of the shared memory block and its sizes. In this example, the ghost cell offset is 1 on both sides of the plane. Within the kernel, we define the tile sizes as constants with the `#define` keyword.

We initialize the shared memory block by loading data from global memory. Depending on the number of planes designated for the variable, we load one or three planes, as shown in Table 5.5. Each thread loads a value in a single plane into the center point. For the case of one plane, the translator generates the assignment at Line 2, which is a load from global memory to shared memory. For the case of three planes, we load the top, center and bottom planes into shared memory (Lines 5-7). If the register optimizer is also turned on, then Mint loads the center point into a register first and then to shared memory (Line 9-13).

[4]The `__device__` keyword is optional in CUDA

Table 5.5: Initialization of shared memory.

```
1  //For the 1-plane case
2  _sh_U[_idy][_idx] = U[_index3D];
3  ...
4  //For the 3-plane case
5  _sh_U[0][_idy][_idx] = U[_index3D - surface]; //load top
6  _sh_U[1][_idy][_idx] = U[_index3D]; //load center
7  _sh_U[2][_idy][_idx] = U[_index3D + surface]; //load bottom
8
9  //if the register flag is on
10 _rU = U[_index3D];
11 _sh_U[_idy][_idx] = _rU; //1-plane
12 ...
13 _sh_U[1][_idy][_idx] = _rU //3-planes, load center
```

5.7.2 Handling Ghost Cells

One of the issues in using shared memory is that it requires special handling for ghost cells. As discussed previously, we assign certain threads the responsibility for loading ghost cells as well as performing computation. Thus, some threads remain idle during ghost cell load. When we find shareable references, we keep a set of off-center references that will require a ghost cell load. We delegate the threads at the boundaries of a tile to load the ghost region. Using an if-statement, we check if the thread ID is mapped to the boundaries of a tile. For example, the threads assigned to the first row of the tile are responsible for loading the north ghost cells of the tile. The threads assigned to the last column of the tile are responsible for east ghost cells. A corner ghost cell will be loaded by a single thread that is closest to the corner. Fig. 5.9 illustrates two threads and their respective ghost cell assignments.

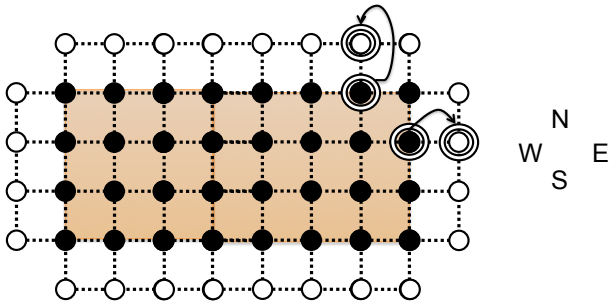


Figure 5.9: Two threads and their respective ghost cell assignments.

We categorize ghost cells according to their offset so that loads for the same category

can be grouped together into the same if-body. Loads for the same category are handled by the same group of threads, avoiding the need to have several back to back if-statements guided by the same conditional. In 2D, a ghost cell might be in the east, west, south or north side of a tile. In addition, there are 4 corners of the tile, totaling 8 different categories. In 3D, there are 26 different patterns of ghost cells; 8 in each of the center, top and bottom planes and two ghost cell loads for the top and bottom center points. Mint's categorization strategy simplifies ghost cell assignments to the threads when we have more sophisticated ghost cell regions such as asymmetric stencils, or stencils with multiple layers of ghost cells.

After loading ghost cell, we synchronize threads with `__syncthreads()` to ensure that all threads in the thread block have finished loading data in shared memory. The synchronization point guarantees the memory consistency among threads within the same thread block.

5.7.3 Replacing Global Memory References

After synchronizing threads, we can replace all the shareable global memory references with their shared memory counterparts. Depending on the number of planes designated for the variable, the shared memory reference changes. If the variable uses one plane in shared memory, then a global memory reference in the form of $[k][j \pm c_j][i \pm c_i]$ becomes $[\_idy \pm c_j][\_idx \pm c_i]$. For example, $U[k][j-1][i+2]$ will become $\_sh\_U[\_idy-1][\_idx+2]$. If the variable uses three planes, then the index in the slowest varying dimension becomes 0, 1 or 2 if its reference is $k-1, k$, or $k+1$ respectively. For example, $U[k-1][j-1][i+2]$ will become $\_sh\_U[0][\_idy-1][\_idx+2]$. Lastly, after synchronizing with other threads in the same thread block, we write the value in shared memory back to global memory unless the variable is read-only.

5.7.4 Shared Memory Code Example

Table 5.6 shows code generated by Mint when the register and shared memory optimizers are both turned on. In Table 5.6, Lines 5 through 16 load ghost cells from global memory into shared memory. If-statements check if the thread is responsible for handling ghost cells. The threads that are at the boundaries of the tile or the grid load the ghost cells. We synchronize at Line 21 to make sure that all loads to the shared memory are completed. At Lines 22-25, the computation makes all the references through registers or shared memory and makes no reference to global memory. `E` and `R` mainly benefit from registers and `Eprev` takes advantage of shared memory because of its stencil access pattern.

Table 5.6: Mint-generated code when both the register and shared memory optimizers are turned on. The input code to the Mint translator is the Aliev-Panfilov model presented in Table 7.10.

```
#define TILE_X 8
#define TILE_Y 8
__device__ __shared__ float _sh_E_prev[TILE_Y + 2][TILE_X + 2];

if (_idx == 1) { //east ghost cells
  _sh_E_prev[_idy][_idx - 1] = E_prev[_index2D - 1];
 }
if (_idx == TILE_X || _gidx == N) { //west ghost cells
  _sh_E_prev[_idy][_idx + 1] = E_prev[_index2D + 1];
 }
if (_idy == 1) { //north ghost cells
  _sh_E_prev[_idy - 1][_idx] = E_prev[_index2D - _width];
 }
if (_idy == TILE_Y || _gidy == N) { //south ghost cells
  _sh_E_prev[_idy + 1][_idx] = E_prev[_index2D + _width];
 }
float _rE_prev = E_prev[_index2D];
_sh_E_prev[_idy][_idx] = _rE_prev;
float _rR = R[_index2D];
float _rE;
__syncthreads();
_rE = _rE_prev + alpha * (_sh_E_prev[_idy + 1][_idx] + _sh_E_prev[_idy - 1][_idx]
               - 4 * _rE_prev + _sh_E_prev[_idy][_idx + 1] + _sh_E_prev[_idy][_idx - 1]);
_rE =_rE - (dt * ((((kk * _rE) * (_rE - a)) * (_rE - 1)) + (_rE * _rR)));
_rR =_rR + dt* (epsilon + ((M1 * _rR) / (_rE + M2))) * (-_rR - ((kk * _rE) * (_rE -b-1)));

E[_index2D] = _rE;
R[_index2D] = _rR;
```

Table 5.7: Part of Mint-generated code when both the register and shared memory optimizers are turned on and chunksize clause is used. We omitted some of the details for the sake of clarity. The input code to the Mint translator is the 3D heat solver presented in Table 2.2.

```
1  double _top_rUold = Uold[_index3D - _slice]; //top
2  double _rUold = Uold[_index3D]; //center
3  ...
4  for (_gidz = _gidz; _gidz <= _upper_gidz; _gidz += 1) {
5      _index3D = _gidx + _gidy * _width + _gidz * _slice; //compute new index
6      _sh_Uold[_idy][_idx] = _rUold;
7      ...
8      //ghost cell loads for center plane
9      ...
10     __syncthreads();
11     double _bottom_rUold = Uold[_index3D + _slice];
12     double _rUnew = c0 * r_Uold +
13                     c1 * (_sh_Uold[_idy][_idx - 1] + _sh_Uold[_idy][_idx + 1] +
14                           _sh_Uold[_idy - 1][_idx] + _sh_Uold[_idy + 1][_idx] +
15                           _top_rUold + _bottom_rUold);
16     Unew[_index3D] = _rUnew;
17     _top_rUold = _rUold;
18     _rUold = _bottom_rUold;
19     __syncthreads();
20 }
```

5.8 Chunksize Clause

As described in the hand implementation in Section 5.1, we can further improve reuse by employing the `chunksize` clause in tandem with on-chip memory optimizations. The effect is to assign each CUDA thread more than one point in the iteration space of the loop nest, enabling shared memory values to be shared by threads in updating adjacent points. This optimization is particularly helpful for 3D stencils as it allows the reuse of data already in shared memory, by chunking along the z-dimension. The programmer can explicitly trigger this optimization by setting a chunking factor in the z-dimension (the 3^{rd} argument of `chunksize` clause). A plane that has been read from global memory starts as the bottom plane, continues as the center plane and then migrates to the top, as depicted in Fig. 5.4. Depending on which compiler flags are enabled, Mint implements the optimization with registers, or shared memory, or both.

Table 5.7 shows a part of the generated code that implements chunking, when both shared memory and registers are used. We iterate over the z-dimension with the loop at Line 4. The number of iterations depends on the chunksize value set by the user. Before we enter the

Table 5.8: Swapping index variables.

```
1  int _top = 0;
2  int _ctr = 1;
3  int _bottom = 2;
4  foreach plane in chunksize
5      ... _sh_Uold[_top][][] + _sh_Uold[_ctr][][] + _sh_Uold[_bottom][][] ...
6      int _tmp = _down;
7      _down = _up;
8      _up = _ctr;
9      _ctr = _down;
10 end of foreach
```

loop, we load the top and center points into registers. Line 6 recomputes the new index for a thread to locate its work assignment. Line 7 copies the register content into shared memory, so that it is visible to other threads in the thread block. We proceed with ghost cell loads into shared memory. After we synchronize with other threads, we read the bottom value from global memory into a register (Line 11). Line 12 reading performs the 7-point stencil computation. The computation reads off-center points from shared memory (the references made to `_sh_Uold)`, and the center, top and bottom points from registers (`r_Uold, _top_rUold, _bottom_rUold`). Lines 17 and 18 rotate the content of the registers: the top becomes the center, while the center becomes the bottom. At the next iteration, we read the new value into the bottom. Finally, we synchronize to ensure that the threads finish reading from shared memory before we write into it again at the next iteration.

The code handling ghost cell loads differs from kernel to kernel depending on whether the kernel requires all three planes in shared memory. The example above uses only one plane and the ghost cells for the center plane are loaded inside the loop. A three-plane implementation would need the ghost cells for top and center to be loaded outside the loop. It would load the ghost cells for the bottom plane inside the loop. For such kernels we also need to rotate the planes in shared memory. Instead of rotating the shared memory content, we define three index variables, “top, ctr and bottom" to refer to the z-dimension of the shared memory block. We swap these index variables, as shown in Table 5.8, to create the same effect of rotating the shared memory content. This is less costly than copying the shared memory blocks.

In addition to the z-dimension, it is possible to employ chunking in other dimensions. In some kernels, we have observed some benefit of assigning more than one elements to a thread in the y-dimension in our hand-written versions. Chunking in the x-dimension is not benefi-

cial because it breaks inter-thread locality. Although the Mint interface allows a user to set a chunking factor in all three dimensions (`chunksize`(c_x, c_y, c_z)), we have not yet implemented the optimization for x and y-dimensions in Mint.

5.9 Miscellaneous

Mint maintains separate index variables to address different arrays even when their sizes in all dimensions are the same. A manual implementation would use the same variable to index the arrays. We introduce another compiler flag, called *-useSameIndex*, to eliminate the need to define separate variables to index arrays. The aim of the flag is to reduce the number of registers allocated to a kernel. Without this flag, the compiler uses separate *width, surface* and *index* variables for two different arrays. When the flag is enabled, the compiler use single variables for *width, surface* and *index*. This flag should be used with caution, because Mint does not verify if array sizes are the same. It relies on the programmer and assumes it is the case.

5.10 Summary

In this chapter, we unveiled the details of the Mint optimizer. First we gave a background on how a stencil kernel is optimized for a GPU system. In the hand-coded implementations, we utilize registers and shared memory to implement chunking (or sliding window) optimization, which eliminates all redundant memory accesses to global memory. Then, we explained how we incorporated these optimizations into the compiler.

The indispensable part of the Mint optimizer is the stencil analyzer. The stencil analyzer collects all array reference expressions and groups them by array name. Among these references, it identifies shareable references that fall into the stencil pattern. In other words, these are the references that Mint considers as “nearest" neighbors. The shareable references are further categorized depending on the position with respect to the center point. This information is used to sort the access frequencies of the arrays and select variables that are amenable to shared memory and register optimizations.

The variables selected by the stencil analyzers are passed to the optimizers to perform register and shared memory optimizations. The optimizer places frequently accesses array references into registers. Since the content of a register is visible to one thread only, the register optimizer enhances access to the central point of a stencil, but not the neighboring points shared by other threads. For that purpose, we implemented a shared memory optimizer which optimizes

the kernel for shared memory. The generated code handles ghost cell loads as well, which might be very complicated if implemented by hand.

The optimizer relies on the stencil analyzer to detect the stencil pattern in the code. The analyzer looks at the index expressions that appear in the array subscripts. These references should be explicitly dependent on the loop indices. To handle more complex subscripts, (e.g., relative index expressions), we need to implement more compiler analysis. Advance analysis will helps us cover a wider spectrum of codes.

Acknowledgements

This chapter, in part, is currently being prepared for submission for publication with Xing Cai and Scott B. Baden. I am the primary investigator and author of the material.

Chapter 6

Commonly Used Stencil Kernels

This chapter demonstrates the effectiveness of the Mint translator by studying a set of widely used stencil kernels in two and three dimensions. The kernels were chosen because of their different memory access patterns and computational intensity. The chapter first provides the background on the computer testbeds and software used throughout this thesis, then presents the performance results for commonly used stencil kernels.

6.1 Testbeds

In this dissertation, to conduct our performance studies, we used a multicore cluster, *Triton*, and two different GPU devices.

6.1.1 Triton Compute Cluster

The Triton cluster [Tri], located at the San Diego Supercomputer Center, is based on the Intel Nehalem processor microarchitecture. A node on Triton contains a dual-socket quad-core Intel Xeon E5530 (Gainestown, 2.40 GHz) processor, totaling 8 processing cores per node. A core is capable of running two hardware threads. Each of the 256 nodes has 24 GB of memory and an 8 MB L3 cache shared by all of the cores in a socket. Each core has a private 64 KB L1 cache (32 KB data and 32 KB instruction) and private 256 KB unified L2 cache. The nodes are connected with a 10-gigabit Myrinet interconnection, giving the system a 256 GB/s bandwidth capacity. According to the stream triad benchmark [McC95], a single thread on a node sustains 10 GB/s memory bandwidth, and 16 threads sustain nearly 25 GB/s (peak of 32 GB/s). A single core has a peak theoretical floating point performance of 9.6 Gflop/s.

Table 6.1: Device Specifications. SM: Stream Multiprocessor

Device Specifications	Tesla C1060 200-series	Fermi Tesla C2050 400-series
Number of SMs	30	14
Number of Cores	240	448
Device Capability	1.3	2.0
Clock Rate (GHz)	1.3	1.15
Memory Size (GB)	4	3
Max Threads/Block	512	1024
Registers/SM	16K	32K
Shared Memory/SM	16KB	48KB
Nvcc Version	4	3.2

The serial and OpenMP programs running on this machine were compiled using the Portland Group compiler *pgCC 10.5-0* with command line flags *-fastsse -mp*. The MPI Fortran programs were compiled using the Portland Group compiler *mpif90 v10.5* with options *-O3 -fastsse -Mflushz -Mdaz -Mnontemporal -Mextend -Mfixed*.

6.1.2 GPU Devices

We employed two GPU devices for our performance studies. The first is a 200-series Tesla C1060 with 4 GB device memory. The device is 1.3 capable and has 30 stream multiprocessors, each with eight 1.3 GHz cores. Each stream multiprocessor has a 16KB shared memory and 64KB of registers. The second GPU device is a 400-series Tesla C2050 based on Fermi architecture. This Fermi device is 2.0 capable and has 3 GB memory. It contains 14 streaming multiprocessors with 32 cores, a total of 448 cores running at 1.15 GHz. Table-6.1 summarizes the specifications of both devices. On the C1060, the CUDA codes were compiled using the Nvidia CUDA compiler *nvcc 4.0* with *-O3* optimization. On the C2050, the codes were compiled using the Nvidia CUDA compiler *nvcc 3.2* with *-O4 -dlcm=ca* optimization.

Table-6.2 highlights the performance of the devices for various metrics. The table lists the peak and sustained floating point rate as well as the memory bandwidth for both devices. We measured the sustained performance numbers using the Shoc-1.1.0 benchmark [DMM+10] developed at the Oak Ridge National Laboratory. The C2050 achieved 121.5 GB/s, or 89% of

Table 6.2: Device Performance, SP: Single Precision, DP: Double Precision, BW: Bandwidth

Performance	Tesla C1060	Tesla C2050
Peak SP (Gflop/s)	622	1030.4
Peak DP (Gflop/s)	78	515.2
Sustained SP (Gflop/s)	409.1	998.3
Sustained DP (Gflop/s)	77.4	501.9
Sustained/Peak SP (Gflop/s)	66%	97%
Sustained/Peak DP (Gflop/s)	99%	97%
DRAM BW (GB/s)	102	144
Sustained Read BW (GB/s)	76.6	127.9
Sustained Write BW (GB/s)	69.0	121.5
Percentage BW Sustained/Peak	75%	89%
Flop:Word Ratio Single (Peak)	24.4	28.6
Flop:Word Ratio Single (Sustained)	21.4	31.2
Flop:Word Ratio Double (Peak)	6.1	28.6
Flop:Word Ratio Double (Sustained)	8.1	31.4

the device's peak theoretical bandwidth. On the other hand, the C1060, which has a slower memory bus, achieved only 76.6 GB/s, or 75% of the theoretical bandwidth.

A significant difference between the devices we employed is double precision performance. Compared to the 200-series of GPUs, Fermi significantly improves the double precision floating point rate. A Fermi stream multiprocessor (32 cores) performs up to 16 double precision fused multiply-add operations per clock. Thus, the double precision arithmetic is half the speed of the single precision. On the 200-series, the double precision arithmetic performs at the 1/8th the speed of the single precision.

Compared to the 200-series of GPUs, the Fermi processor almost doubles the number of cores on the processor. However, the number of registers per core drops by half. The implication is that if we increase the number of cores but not the register storage, then the compute-bound kernels may not be able to take advantage of the added cores. The memory bandwidth-bound case is more complicated due to the new on-chip memory design of Fermi. Fermi's 64 KB on-chip memory can be configured as 48 KB shared memory and 16 KB L1 cache or the other way around. Unless stated otherwise, we used the default configuration of on-chip memory: 48KB shared memory and 16KB L1.

The table also lists the *flops:word* ratio of the *devices* based on the peak and sustained performance numbers. The flops:word ratio for an *application* is the ratio between the number of floating point operations and the number of memory accesses performed per element and is an important determinant of performance. If an application's flops:word ratio is below the flops:word ratio of the device, then the application is bandwidth-limited on that device otherwise it is compute-bound. Table 6.2 clearly shows that the GPU devices have higher flop:word ratios compared to that of the stencil kernels (discussed shortly). This indicates that the performance of the stencil applications is more likely to depend on the device memory bandwidth.

Table 6.3: A summary of stencil kernels. The $\pm$ notation is short hand to save space, $u^n_{i\pm1,j} = u^n_{i-1,j} + u^n_{i+1,j}$. The 19-pt stencil Gflop/s rate is calculated based on the reduced flop counts which is 15 (see Section 5.1.3 for details).

Stencil kernel	Mathematical description	In,Out arrays	Read,Write per point	Operations per point
2D Heat 5-point	$u^{n+1}_{i,j} = c_0 u^n_{i,j} + c_1 \left(u^n_{i\pm1,j} + u^n_{i,j\pm1} \right)$	1,1	5,1	2(∗),4(+)
3D Heat 7-point	$u^{n+1}_{i,j,k} = c_0 u^n_{i,j,k} + c_1 \left(u^n_{i\pm1,j,k} + u^n_{i,j\pm1,k} + u^n_{i,j,k\pm1} \right)$	1,1	7,1	2(∗),6(+)
3D Poisson 7-point	$u^{n+1}_{i,j,k} = c_0 b_{i,j,k} + c_1 \left(u^n_{i\pm1,j,k} + u^n_{i,j\pm1,k} + u^n_{i,j,k\pm1} \right)$	2,1	7,1	2(∗),6(+)
3D Heat 7-point variable coefficient	$u^{n+1}_{i,j,k} = u^n_{i,j,k} + b_{i,j,k} + c \Big[\kappa_{i+\frac{1}{2},j,k} \left(u^n_{i+1,j,k} - u^n_{i,j,k} \right) - \kappa_{i-\frac{1}{2},j,k} \left(u^n_{i,j,k} - u^n_{i-1,j,k} \right) + \kappa_{i,j+\frac{1}{2},k} \left(u^n_{i,j+1,k} - u^n_{i,j,k} \right) - \kappa_{i,j-\frac{1}{2},k} \left(u^n_{i,j,k} - u^n_{i,j-1,k} \right) + \kappa_{i,j,k+\frac{1}{2}} \left(u^n_{i,j,k+1} - u^n_{i,j,k} \right) - \kappa_{i,j,k-\frac{1}{2}} \left(u^n_{i,j,k} - u^n_{i,j,k-1} \right) \Big]$	3,1	15,1	7(∗),19(+),
3D Poisson 19-point	$u^{n+1}_{i,j,k} = c_0 \Big[b_{i,j,k} + c_1 \left(u^n_{i\pm1,j,k} + u^n_{i,j\pm1,k} + u^n_{i,j,k\pm1} \right) + u^n_{i\pm1,j\pm1,k} + u^n_{i\pm1,j,k\pm1} + u^n_{i,j\pm1,k\pm1} \Big]$	2,1	19,1	2(∗),18(+)

6.2 Commonly Used Stencil Kernels

To demonstrate the effectiveness of the Mint translator, we used a set of widely used stencil kernels in two and three dimensions. We included two of the most well known stencils: the 5-point stencil approximation of the 2D Laplacian operator in connection with explicitly

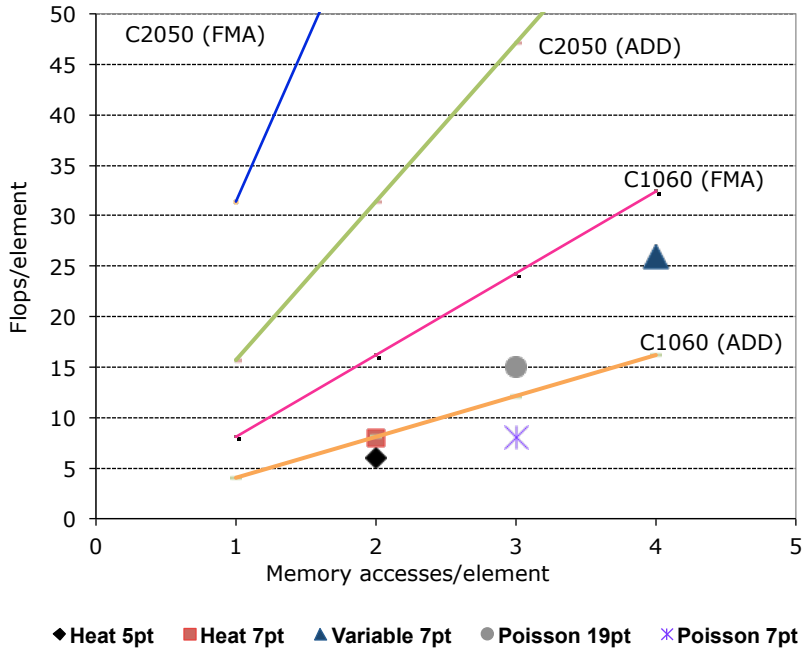


Figure 6.1: Flops/element and memory accesses/element of the kernels and flops:word ratios for the test devices.

solving a 2D heat equation that has no source term and the corresponding 7-point stencil in 3D. In addition, we considered 7-point and 19-point Poisson solvers. The Poisson solvers introduces an additional memory stream. Lastly, we looked at the 7-point finite difference stencil for explicitly solving a 3D heat equation with a variable coefficient plus a source term. Table 6.3 summarizes the characteristics of each kernel. The superscript n denotes the discrete time step number (an iteration), the subscripts (i, j, k) denotes the spatial index.

The stencil kernels exhibit different *flops:word* ratios. The lowest ratio is 3 and the highest is 6.5. Fig. 6.1 plots the flop and memory access relationship for each kernel. It also shows the double precision flops:word ratio for the devices that we used. Since our kernels contain a mix of adds, multiplies and fused-multiply-adds (FMA)[1], the true peak throughput of a device depends on the mix of instruction types that it executes. Thus, we plotted 2 curves for each processor: one for FMA, which is the sustained double precision flop rate, and one for adds and multiplies, which run at half the rate of FMA. The actual throughput for a stencil kernel lies between these curves. In order to compute the ratio for the devices we use empirical performance

[1] A fused-multiply-add is a floating-point multiply-add operation $(a + b * c)$ performed in one instruction, which is more accurate than performing the operation in two separate instructions.

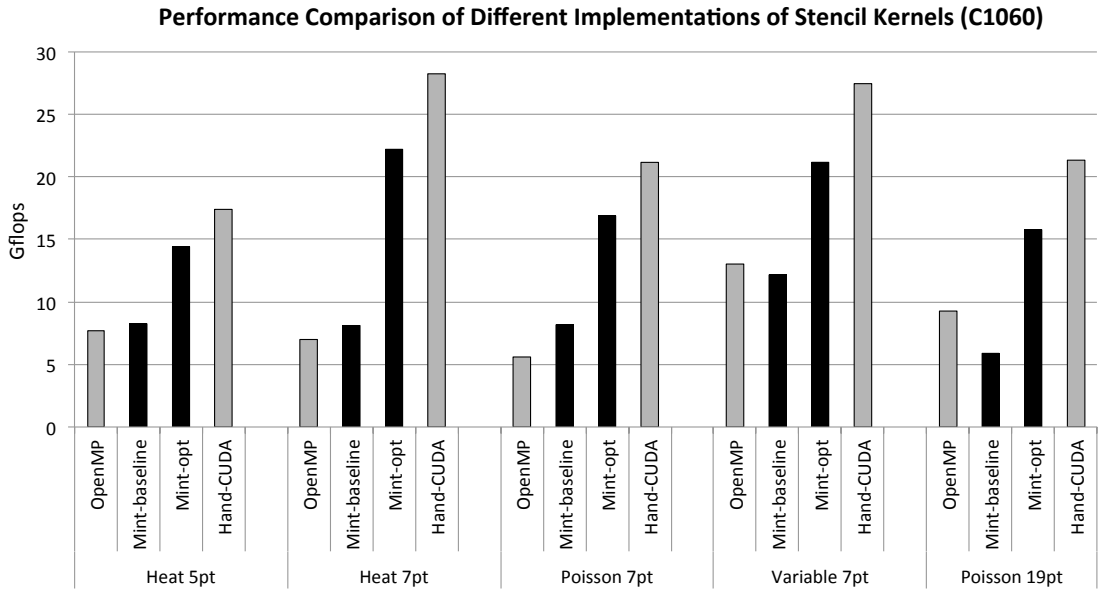


Figure 6.2: Performance comparison of the kernels. OpenMP ran with 8 threads on the E5530 Nehalem. Mint-baseline corresponds to the Mint baseline translation without using the Mint optimizer, Mint-opt with optimizations turned on, and Hand-CUDA is hand-optimized CUDA. The Y-axis shows the measured Gflop rate. Heat 5-pt is a 2D kernel, the rest are 3D.

values from Table 6.2 rather than the theoretical peak performance values because they are more realistic. A kernel whose ratio is below the flop:word ratio of the device is bandwidth-limited on that device otherwise it is compute-bound. The figure clearly shows that the stencil kernels are expected to be bandwidth-limited on both devices. The figure also shows that the 400-series of GPUs drastically improved the floating point rate over the 200-series. However the memory bandwidth did not increase proportionally.

6.2.1 Performance Comparison

We compared the performance of Mint-generated CUDA with hand-written CUDA and with OpenMP. All computations were run in double precision. Fig. 6.2 compares the performance of the stencil kernels and their 4 different implementations. *OpenMP* results were obtained by running 8 threads on one Triton node of 8 Nehalem cores. The remaining three versions ran on the Tesla C1060. *Mint-baseline* is the result of compiling Mint-annotated C source without enabling any Mint optimizations, and *Mint-opt* reports the best performance as a result of

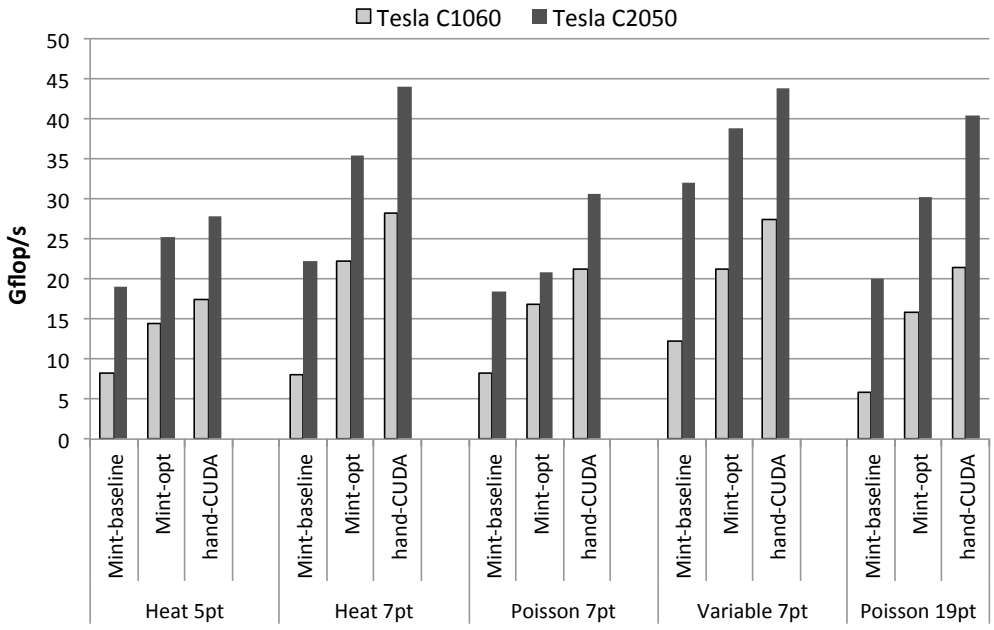


Figure 6.3: Performance comparison of the Tesla C1060 and C2050 on the stencil kernels. Mint-baseline corresponds to the Mint baseline translation without using the Mint optimizer, Mint-opt with optimizations turned on, and Hand-CUDA is hand-optimized CUDA. The Y-axis shows the measured Gflop rate. Heat 5-pt is a 2D kernel, the rest are 3D.

the Mint optimizations. All the Mint-generated CUDA codes were compiled with `nvcc` without any modification. Lastly, *Hand-CUDA* refers to manually implemented and optimized CUDA C. Mint was **not** used to produce this code. The Mint optimizer improves the performance of the baseline Mint, achieving between 78% and 83% of the performance of the aggressively hand-optimized CUDA versions. The optimizer delivers 1.6 to 3.2 times the performance of OpenMP running with 8 threads. We analyze the results more in depth in Sec 6.2.2.

Fig. 6.3 compares the performance of the C1060 with that of the C2050. On average, the C2050 shows 1.66x and 1.62x performance advantage over the C1060 for both the Mint-opt and hand-CUDA respectively. The results are in line with the ratio between the two devices' sustained memory bandwidth (which is 1.67x). On the other hand, the C2050 shows a 2.3x to 3.4x performance advantage over the C1060 for the baseline code without performing any optimizations. This is due to the fact that there is an L1 cache on the C2050, which benefits the naive implementations. Overall on the C2050, Mint achieved am average pf 81% ranging from 68% to 90% of the performance of the hand-written CUDA.

In the next section we explain the performance tuning process for both devices and how the the *Mint-opt* results were achieved.

6.2.2 Compiler-Assisted Performance Tuning

In order to tune kernel performance, the Mint interface provides tunable for-loop clauses and the Mint optimizer provides selectable optimizations. We present the performance of the stencil kernels based on 3 configurations: *opt-1, opt-2 and opt-3*.

- *Opt-1* turns on shared memory optimization (*-shared*).
- *Opt-2* utilizes the `chunksize` clause and *-shared*.
- *Opt-3* adds register optimizations (*-shared -register*).

The optimizations are cumulative, thus opt-3 includes all three optimizations. In all Mint code, loops are annotated with `nest(all)`. In 3D, we set `tile` sizes as follows, all resulting in 16x16 thread blocks, except for the 2D Heat kernel, which uses a 16x16 tile size.

- *baseline* and *opt-1* use 2D tiles `tile(16,16,1)` and `chunksize(1,1,1)`. Each thread computes just one element.
- *opt-2* and *opt-3* use 3D tiles `tile(16,16,64)` and `chunksize(1,1,64)`. Each thread computes 64 elements in the z-dimension.

The best choice of chunksize and tile size depends on the application and device. The programmer has to experiment with different configurations. Our recommendation is to choose a tile size multiple of 16 in the x-dimension to ensure the aligned memory accesses. The tile size in the y-dimension can be chosen smaller than 16 depending on the on-chip memory requirement of the kernel. Fewer threads mean more registers per thread, thus more thread blocks per stream multiprocessor. The value of `chunksize` is less sensitive to the on-chip resources but chunksize clause itself will increase the register usage by the kernel. We recommend a value multiple of 32 because if the chunksize is too small, it won't amortize the overhead introduced by the loop.

Fig. 6.4 shows the performance impact of the different optimizations on the Tesla C1060. We report performance as a speedup over Mint-generated code without any of the compiler optimization turned on (baseline). Generally, performance improves with the level of optimization, though not in all cases. Not all optimizations are relevant in two dimensions, for example, chunking in the z-dimension (*opt-2*). Shared memory optimization (*opt-1*) is always helpful. It takes

advantage of the significant opportunities for memory re-use in stencil kernels, reducing global memory traffic significantly. We observed performance improvements in the range of 30-70%.

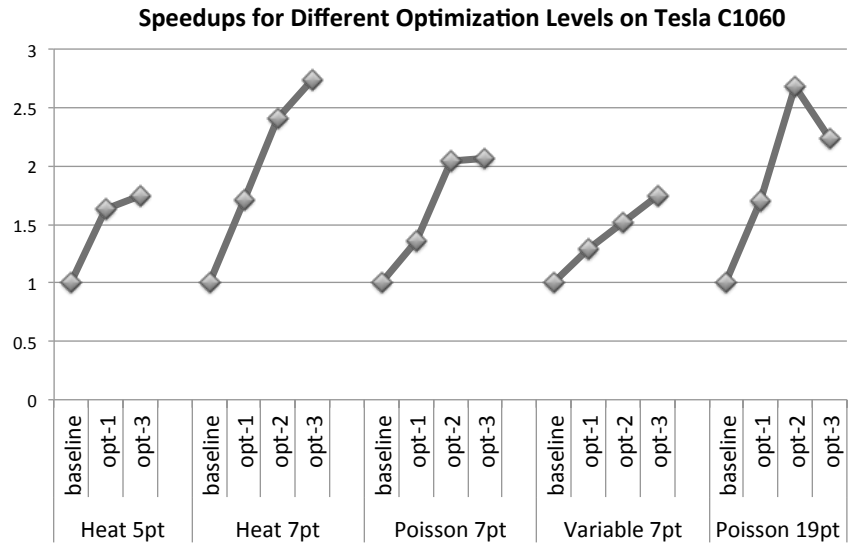


Figure 6.4: Effect of the Mint optimizer on the Tesla C1060. The baseline resolves all the array references through device memory. *Opt-1* turns on shared memory optimization *(-shared)*. *Opt-2* utilizes the chunksize clause and *-shared*. *Opt-3* adds register optimizations *(-shared -register)*.

The speedups attributed to tiling for shared memory differ according to the number of data streams that exhibit reuse. The Poisson 7-pt and Heat 7-pt stencils have the same flop counts but Poisson 7-pt requires an additional input grid, increasing the number of data streams from 2 to 3. As a result, its performance lags behind the Heat 7-pt because one of the input arrays (the right hand side) does not exhibit data reuse and cannot benefit from shared memory. On the other hand, the 7-pt variable coefficient kernel (Variable 7-pt) has 4 data streams, and 2 of the streams exhibit reuse. The Mint optimizer places the corresponding two meshes in shared memory, trading off occupancy for reduced global memory traffic. More shared memory usage means fewer concurrent thread blocks, thus lower occupancy. Since Variable 7-pt uses twice the shared memory as Heat 7-pt, occupancy is cut in half, and the performance improvement is modest. We found that if we allowed only one of the two data streams in question to reside in shared memory *(-shared=1)*, then overall performance increased by 45% compared to when we used no shared memory. When we put both data streams in shared memory by setting the compiler flag *-shared=2*, the overall improvement increased to 85%. Since other stencil kernels have only 1 data stream exhibiting reuse, we did not experiment on those with different shared

memory options.

The *opt-2* flag applies window sliding via the `chunksize` clause on top of the shared memory optimization (refer to Section 5.8 for details). In *opt-1*, threads can share data (reads) within the xy-plane only, whereas *opt-2* assigns each thread a column of values, allowing reuse in the z-dimension as well. Thus, the thread block is responsible for 3D tile of data and can share the work across xy-planes. The generated kernels perform only the necessary loads and stores to the device memory for the inner points. Once an element is read, no other thread will load the value, except for the ghost cells. This optimization has a significant impact on performance. The effect is particularly pronounced for the Poisson 19-pt stencil, owing to the high degree of sharing between threads.

The *opt-3* flag implements register optimizations on top of *opt-2*. This optimization helps in two ways: (1) an instruction executes more quickly if its operands are in registers [VD08] and (2) registers augment shared memory with plentiful on-chip storage (64KB compared with 16KB of shared memory). We can store more information on-chip and reduce pressure on shared memory. By reducing pressure on shared memory, the device can execute more thread blocks concurrently, i.e. with higher occupancy. Indeed, the register optimization improves performance in nearly all the kernels. The one exception is the 19-point stencil. This is an artifact of the current state of our optimizer. The hand-CUDA version of the 19-point kernel uses registers more effectively. It eliminates common expressions appearing in multiple slices of the input grid. It also uses registers to store intermediate sums along edges, and reuses the computed sums in multiple slices. The effect is to reduce the number of flops performed per data point, discussed in Section 5.1.3. The opportunity does not arise in the 7-pt stencils because of the limited sharing: only one value is used from the top and bottom slices. Future work will implement this optimization in the compiler and we expect to increase the performance of the 19-pt stencil further in order to close the performance gap with the hand-coded CUDA.

Fig. 6.5 shows the speedups for different optimization levels on the C2050. We experimented with two configurations with L1 cache: *L1 > Shared* reports the results when we favor larger cache with the *-preferL1* flag. That is, on-chip memory is configured as 48KB of L1 cache and 16KB of shared memory. *Shared > L1* indicates that on-chip memory is configured as 48KB of shared memory and 16KB of L1 cache. The Fermi Tuning Guide [Nvi10b] suggests experimentation to determine the best cache configuration for a given kernel. We got some insight into when preferring a larger L1 is more advantageous, though more analysis is needed to comprehend the Fermi architecture. We expect that employing a larger cache benefits the baseline

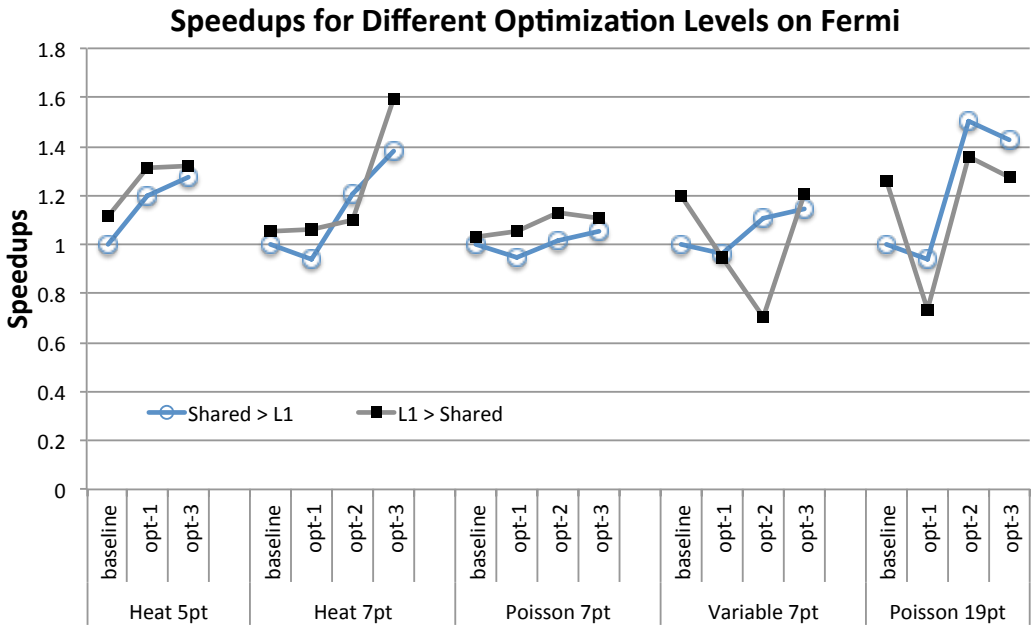


Figure 6.5: Effect of the Mint optimizer on the performance on the Tesla C2050. The baseline resolves all the array references through device memory. *L1 > Shared* favors larger cache. *Shared > L1* favors larger shared memory. *Opt-1* turns on shared memory optimization *(-shared)*. *Opt-2* utilizes the chunksize clause and *-shared*. *Opt-3* adds register optimizations *(-shared -register)*

variant because L1 can cache the global memory accesses. We can observe this trend in the figure. Another case where L1 cache should be favored is when the kernel requires a small amount of shared memory. In other words, if the 16KB shared memory is sufficient to buffer the stencil arrays, a larger L1 cache should be used so that non-stencil arrays, for example coefficient arrays, can be cached. What is sufficient is highly correlated to the device occupancy. If the kernel is not limited by shared memory, but limited by the number of registers or warp size, favoring L1 should improve the performance because the available shared memory suffices.

We used the Nvidia Visual Profiler [Nvi] to understand the effect of the optimizations on the C2050. The profiler revealed that L1 global load hit rates are higher for the kernels that do not use shared memory or that use smaller shared memory. This is because the shared memory accesses are not cached. For example, for the Heat-7pt kernel, the *baseline* variant has a 49.2% L1 load hit rate and 95 GB/s memory throughput. On the other hand, the shared memory variant which explicitly fetches the data into on-chip memory and by-passes the L1 cache has a 16.8% L1 hit rate. The *opt-1* variant yields 90 GB/s throughput, which is slightly lower than that of

baseline. The *opt-1, -preferL1* variant yields 101 GB/s throughput with a 27.5% L1 hit rate. Even though the hit rate is lower that of the *baseline* but higher than *opt-1*, which prefers shared memory. This is because shared memory buffers the stencil array and the cache buffers the non-stencil array.

As we discussed previously, another case where preferring a larger L1 cache should be more advantageous is when the kernel only uses small amount of shared memory. In such a case, reserving 16KB of shared memory is sufficient. For example, *opt-3* applied to the Heat 7-pt makes the same amount of read requests to the global memory regardless of *-preferL1* is used or not. However, when *Shared* > *L1*, the L1 global load hit rate is 85% less than that of *L1* > *Shared*. Both variants require only 2592 bytes of shared memory and the limiting factor for the kernels is the number of registers. As a result, a larger L1 cache is more advantageous because it does not affect the device occupancy but improves the hit rates.

An example where the occupancy is affected by the cache configuration is the *opt-2* in Variable 7-pt in Fig. 6.5. The configuration of *opt-2, L1> Shared* substantially degrades performance because *opt-2* implements the chunking optimization via shared memory. It places two of the data streams into the shared memory, each of which references 3 planes, requiring 16KB of shared memory for a 16x16 tile size. When configured as 48KB shared memory, a stream multiprocessor can concurrently run 3 thread blocks. When configured as 16KB of shared memory, the number of active blocks drops to 1, which explains the performance degradation for the Variable 7-pt. However, *opt-3* for the same kernel uses registers, which reduces the shared memory requirement by 2/3 because the kernel requires only 1 plane in shared memory as opposed to 3. Then, *-preferL1* becomes slightly more advantageous.

A similar issue with occupancy arises in the Poisson 19-pt kernel on the C2050 as well. The kernel prefers more shared memory than L1 cache because it needs 3 planes of data in shared memory. When we use a larger L1 cache, the occupancy drops from 50% to 33%. Using registers does not reduce the shared memory usage for this kernel as opposed to Variable 7-pt or Heat 7-pt because of the stencil pattern of the 19-pt kernel. The kernel makes references to the edges in the top and bottom planes in addition to the center. The big jump in performance from *opt-1* to *opt-2* is that the DRAM reads are cut by half from *opt-1* to *opt-2*. *Opt-2* employs the chunking optimization, which leads to 41 % less dram read requests.

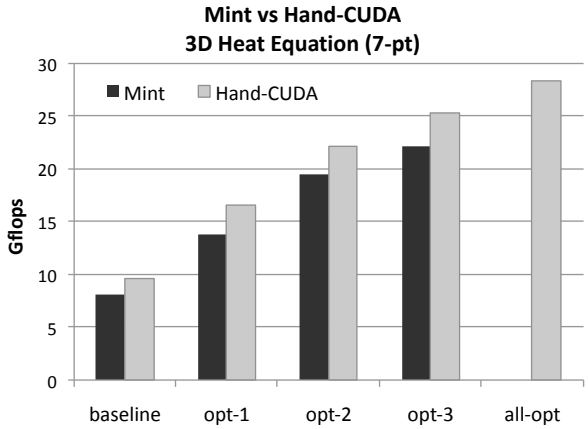


Figure 6.6: Comparing the performance of Mint-generated code and hand-coded CUDA. *All-opt* is the same as the Hand-CUDA variant used in Fig.6.2. *All-opt* indicates additional optimizations on top of Hand-CUDA *opt-3*. The results were obtained on the Tesla C1060.

6.2.3 Mint vs Hand-CUDA

To better understand the source of performance gap between the Mint-generated and hand-optimized CUDA code, we analyze the Heat 7-pt kernel in greater depth on the Tesla C1060. Fig. 6.6 compares the performance of the Mint-generated code for the available optimizations with the hand-coded CUDA (*Hand-CUDA*) that manually implements the same optimization strategies. *All-opt* is the same as the Hand-CUDA variant used in Fig.6.2, which includes additional optimizations that we have not implemented in Mint.

While the Hand-CUDA and Mint variants realize the same optimization strategies, they implement the strategies differently. One way in which the implementations differ is in how they treat padding, which helps ensure that memory accesses coalesce. Mint relies on *cudaMalloc3D* to pad the storage allocation. This function aligns memory to the start of the mesh array, which includes the ghost cells. On the other hand, the *Hand-CUDA* implementation pre-processes the input arrays and pads them to ensure that all the global memory accesses are perfectly aligned. Memory is aligned to the inner region of the input arrays, where the solution is updated. Ghost cells are far less numerous so it pays to align them to the inner region, which accounts for the lion's share of the global memory accesses. We can achieve this effect in the Mint implementation if we pad the arrays manually prior to translation. In so doing, we observed a 10% performance improvement, on average, in the Mint-generated code at all optimization levels. Mint *opt-3* closes the gap from 86% to 90% of the Hand-CUDA *opt-3* with padding.

Mint generates code that uses more registers than the hand-optimized code. This mainly stems from the fact that it maintains separate index variables to address different arrays even when the subscript expressions are shared among references to the different arrays. Combined with manual padding, reduction in index variables improved the performance by 10% and provided us with the same performance for Mint *opt-3* and the Hand-CUDA *opt-3*.

There is also an additional hand coded variant in Fig. 6.6, called *all-opt*, that does not appear in the compiler. This variant supports an optimization we have not included in Mint yet. We built the optimization on top of the *opt-3 HandCuda* variant. Currently, Mint supports `chunksize` for the z-dimension only. The *all-opt* variant implements chunking in y-dimension as well, providing a 12% improvement over Hand-CUDA *opt-3*. It assigns a modest number of elements (2 to 8) in y-dimension to a thread. If implemented in the compiler, for example, the Mint variant will be annotated with `chunksize(1,4,64)`.

6.3 Summary

We have demonstrated that for a set of widely used stencil kernels the source-to-source translator and optimizer of the Mint programming model generates highly optimized CUDA C that is competitive with hand coding. The benefit of our approach is to simplify the view of the hardware while incurring a reasonable abstraction overhead. Most of the kernels were 3-dimensional, where the payoff for successful optimization is high, but so are the difficulties in optimizing CUDA code by hand. In the next Chapter, we will apply Mint to more complex stencil applications solving real-world problems, which are even harder to port to GPUs.

On the Tesla C1060 device, on-chip memory and chunking optimizations are crucial to delivering high performance. The optimizations steadily improve the performance over the *baseline*, delivering speedups in the 1.7-2.7X range. The optimized variants realize 78% to 83% of the performance obtained by hand-optimized CUDA. The results on the C2050 suggests that further analysis is necessary to better target the compiler for the 400-series. The compiler optimizations provided speedups in the range of 1.1-1.6X. Mint achieves 68% to 90% of the hand-CUDA. On the C2050 the performance gain by the optimizer is less predictable because of the presence of L1 cache and requires more tuning. The combination of compiler flags and for-loop directives that yield best performance varies kernel to kernel. Hence, a programmer has to tune a kernel by generating several variants of it. An auto-tuning tool similar to those in [LME09, WOCS11] could alleviate this process, which remains as future work.

Acknowledgements

This chapter, in part, is a reprint of the material as it appears in International Conference on Supercomputing 2011 with the title “Mint: Realizing CUDA performance in 3D Stencil Methods with Annotated C" by Didem Unat, Xing Cai and Scott B. Baden. The dissertation author was the primary investigator and author of this paper.

Chapter 7

Real-World Applications

This chapter presents case studies that we conduct to test the effectiveness of Mint on real-world applications coming from different problem domains. The first application is a cutting-edge seismic modeling application and we collaborated with Jun Zhou and Yifeng Cui at the San Diego Supercomputer Center. The second application comes from computer vision, the Harris interest point detection algorithm, developed by Han Suk Kim and Júrgen Schulze in the Computer Science department at University of California San Diego. In our third study, we investigate the 2D Aliev-Panfilov model that simulates the propagation of electrical signals in cardiac cells. The model was studied by the researchers at the Simula Research Laboratory in Norway and our study is a joint work with Prof. Xing Cai from Simula.

This chapter will go over each application in turn. First we present background for application including the numerical model. For every application, we analyze the most time-consuming loops, the stencil structures, memory access patterns, and arithmetic intensity. Thereafter, we discuss the Mint implementation and the performance of the generated code along with performance tuning. If available, we compare the performance of the Mint generated CUDA with hand-written CUDA implementations of the application. Finally, we close with a few concluding remarks and discuss optimization techniques that remain to be explored in the future.

7.1 AWP-ODC Seismic Modeling

7.1.1 Background

AWP-ODC is a Petascale finite difference anelastic wave propagation code [Ols94] used by researchers at the Southern California Earthquake Center for large-scale wave propagation

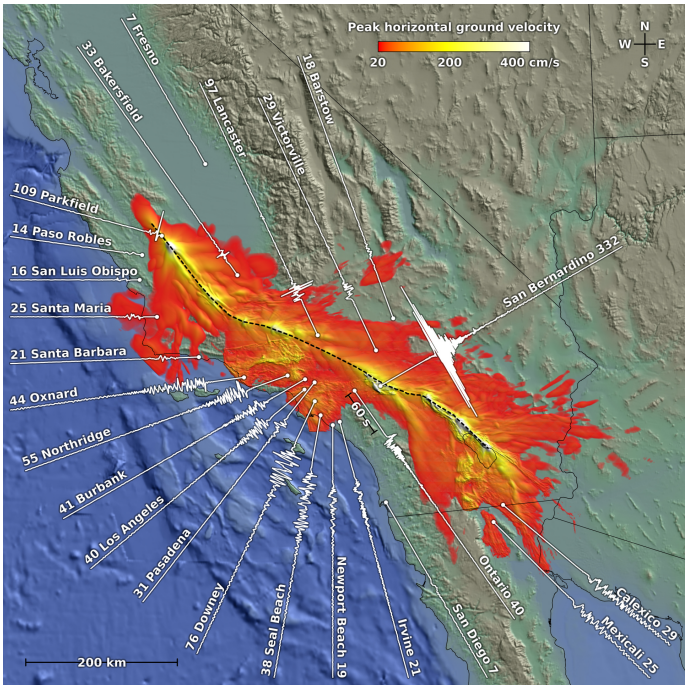


Figure 7.1: The colors on this map of California show the peak ground velocities for a magnitude-8 earthquake simulation. White lines are horizontal-component seismograms at surface sites (white dots). On 436 billion spatial grid points, the largest-ever earthquake simulation, in total 360 seconds of seismic wave excitation up to frequencies of 2 Hz was simulated. Strong shaking of long duration is predicted for the sediment-filled basins in Ventura, Los Angeles, San Bernardino, and the Coachella Valley, caused by a strong coupling between rupture directivity and basin-mode excitation [COJ$^+$10].

simulations, dynamic fault rupture studies and improvement of the structural models. The AWP-ODC model was originally developed by Kim Bak Olsen at University of Utah for wave propagation calculations, with staggered-grid split-node dynamic rupture features added by Steven Day of San Diego State University [DD07]. The code has been validated for a wide range of problems, from simple point sources in a half-space to dipping extended faults in 3D crustal models [ODM$^+$06] and has been extensively optimized at the San Diego Supercomputer Center.

AWP-ODC achieved “Magnitute 8” (M8) status in 2010, a full dynamic simulation of a magnitude-8 earthquake on the southern San Andreas fault, at a maximum frequency resolution of 2 Hz. At the time of writing, this is the largest earthquake ever simulated. The M8 simulation was calculated using 436 billion unknowns, on a uniform grid with a resolution of $40m^3$, and

produced 360Êseconds of wave propagation. The simulation, written in MPI Fortran, sustained 220 Tflop/s for 24 hours on DOE's Cray XT5 Jaguar system [Jag] at the National Center for Computational Sciences, and was an ACM Gordon Bell finalist in 2010 [COJ+10]. Fig. 7.1 shows the simulation result as presented in [COJ+10]. The simulation predicts large amplification with peak ground velocities exceeding 300 cm/s at some locations in the Ventura Basin, and 120 cm/s in the deeper Los Angeles Basin. San Bernardino appears to be the area hardest hit by M8, due to directivity effects coupled with basin amplification and its proximity to the fault.

AWP-ODC has recently been adapted to the reciprocity-based CyberShake simulation [Cyb] project. In the Cybershake project, a state-wide (California) physics-based seismic hazard map is created, based on simulation results accurate up to frequencies of 1 Hz. This project is estimated to consume hundreds of millions of processor-core hours for its more than 4000 site calculations. Each site independently runs the AWP-ODC code twice, one for wave propagation and another for reciprocity-based simulation using different inputs. Current simulations run on mainframes provisioned with conventional processors. Normally each site calculation runs on several hundreds cores for 1 or 2 days. The ultimate goal of SDSC researchers is to employ GPU clusters, or mainframes containing GPUs, in order to reduce the turnaround time of the massive simulation suites (at least 2×4000 independent simulations). Optimizing single GPU performance is a necessary pre-requisite step towards this goal.

7.1.2 The AWP-ODC Model

AWP-ODC solves a coupled system of partial differential equations, using an explicit staggered-grid finite difference method [DD07], fourth-order accurate in space and second-order accurate in time. The code creates estimates of two coupled quantities in space and time. Let $v = (v_x, v_y, v_z)$ denote the velocity of each material particle, and σ denote the symmetric stress tensor for each particle point, where

$$\sigma = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{zx} & \sigma_{zy} & \sigma_{zz} \end{bmatrix} \tag{7.1}$$

The governing elastodynamic equations can be written as

$$\partial_t v = \frac{1}{\rho} \nabla \cdot \sigma \tag{7.2}$$

$$\partial_t \sigma = \lambda (\nabla \cdot v) I + \mu (\nabla v + \nabla v^T) \tag{7.3}$$

where λ and μ are the Lamé elastic constants [AF05] and ρ is the density. By decomposing Eqn. 7.2 and 7.3 component-wise, we get three scalar-valued equations for the velocity vector and six scalar-valued equations for the stress tensor. The velocity equations for v_x, v_y and v_z:

$$\frac{\partial v_x}{\partial t} = \frac{1}{\rho}\left(\frac{\partial \sigma_{xx}}{\partial_x} + \frac{\partial \sigma_{xy}}{\partial_y} + \frac{\partial \sigma_{xz}}{\partial_z}\right) \tag{7.4}$$

$$\frac{\partial v_y}{\partial t} = \frac{1}{\rho}\left(\frac{\partial \sigma_{yy}}{\partial_y} + \frac{\partial \sigma_{xy}}{\partial_x} + \frac{\partial \sigma_{yz}}{\partial_z}\right) \tag{7.5}$$

$$\frac{\partial v_z}{\partial t} = \frac{1}{\rho}\left(\frac{\partial \sigma_{zz}}{\partial_z} + \frac{\partial \sigma_{xz}}{\partial_x} + \frac{\partial \sigma_{yz}}{\partial_y}\right) \tag{7.6}$$

The stress equations:

$$\frac{\partial \sigma_{xx}}{\partial t} = (\lambda + 2\mu)\frac{\partial v_x}{\partial_x} + \lambda\left(\frac{\partial v_y}{\partial_y} + \frac{\partial v_z}{\partial_z}\right) \tag{7.7}$$

$$\frac{\partial \sigma_{yy}}{\partial t} = (\lambda + 2\mu)\frac{\partial v_y}{\partial_y} + \lambda\left(\frac{\partial v_x}{\partial_x} + \frac{\partial v_z}{\partial_z}\right) \tag{7.8}$$

$$\frac{\partial \sigma_{zz}}{\partial t} = (\lambda + 2\mu)\frac{\partial v_z}{\partial_z} + \lambda\left(\frac{\partial v_x}{\partial_x} + \frac{\partial v_y}{\partial_y}\right) \tag{7.9}$$

$$\frac{\partial \sigma_{xy}}{\partial t} = \mu\left(\frac{\partial v_x}{\partial_y} + \frac{\partial v_y}{\partial_x}\right) \tag{7.10}$$

$$\frac{\partial \sigma_{xz}}{\partial t} = \mu\left(\frac{\partial v_x}{\partial_z} + \frac{\partial v_z}{\partial_x}\right) \tag{7.11}$$

$$\frac{\partial \sigma_{yz}}{\partial t} = \mu\left(\frac{\partial v_y}{\partial_z} + \frac{\partial v_z}{\partial_y}\right) \tag{7.12}$$

The Perfectly Matched Layers approach [MO03] is used on the sides and bottom of the grid, and a zero-stress free surface boundary condition is used at the top. The Perfectly Matched Layers approach separates wavefields propagating parallel and perpendicular to the boundary to assure that outgoing waves are absorbed at the interface and not reflected back in the interior.

Table 7.1: Description of 3D grids in the AWP-ODC code. * $r1 - r6$ hold temporal values during computations but they are not outputs.

3D Grids	Component Description	Access Pattern	Input	Output
u1,v1,w1	Vector-valued particle velocity (wave propagation)	13 point stencil	Yes	Yes
xx,yy,zz	Independent stress components for each particle point	13 point stencil	Yes	Yes
xy,xz,yz		13 point stencil	Yes	Yes
r1,r2,r3,r4,r5,r6	Six intermediate variables for stress calculation	1 point	Yes	No*
d1,d2,d3	Density constants for velocity	1 point	Yes	No
xl,xm,xmu1,xmu2,xmu3	Elastic stress constant factors	1 point	Yes	No
h,h1,h2,h3,qpa,vx1,vx2	Material properties of quality factors for stress	1 point	Yes	No
dcrj	Constant for processing wave propagation near the boundary	1 point	Yes	No

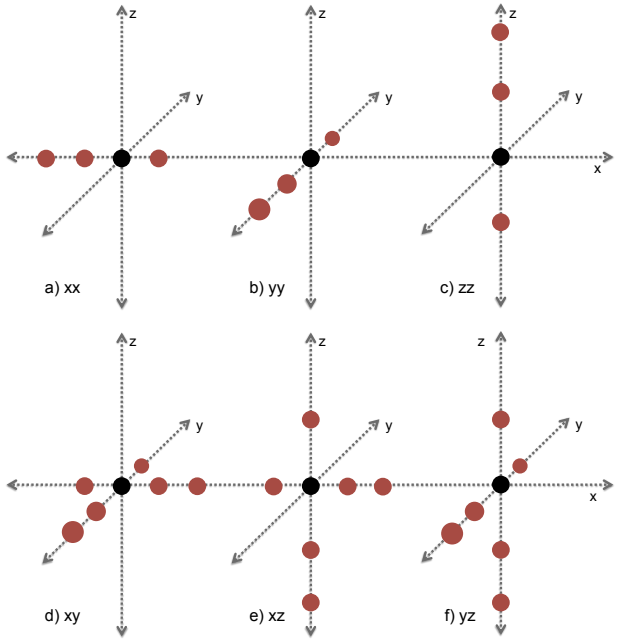


Figure 7.2: Stencil shapes of the stress components used in the velocity kernel. The kernel uses a subset of asymmetric 13-point stencil, coupling 4 points from *xx*, *yy* and *zz* and 8 points from *xy*, *xz* and *yz* with their central point referenced twice.

7.1.3 Stencil Structure and Computational Requirements

An AWP-ODC simulation entails updating two coupled quantities in space and in time: vector-valued velocity (u_1, v_1, w_1), corresponding to (v_x, v_y, v_z) in Eq. 7.2, and the stress σ. The stress tensor is represented on six 3D grids (xx, yy, zz, xy, xz, yz), that correspond to σ_{xx} etc. in Eqn. 7.3. The stress computation uses six additional arrays, $r1$ to $r6$, for temporary computations. Another 16 arrays are needed for spatially-varying coefficients in 3D. These arrays Êparticipate in the computation, but their values remain constant throughout the simulation.

Table 7.1 describes the variables and the corresponding arrays used in the AWP-ODC. In all, the simulation references 31 three-dimensional arrays. Table 7.2 shows the pseudo-code for the main loop. For each iteration, we compute the velocity field in a triply nested loop using an asymmetric 13-point stencil and then update boundary values. We refer to this triple-nested loop as *velocity*. Fig. 7.2 shows the stencil shapes of the input grids to the velocity kernel. The kernel refers to 4 points from the xx, yy and zz grids, and 8 points from xy, xz and yz, with central points referenced twice. The pseudo-code in Table 7.2 shows the calculation of the u_1 velocity component. The others are similar and for the sake of brevity will not be shown. As with the

other velocity components, u_1 is a function of three of the stress components.

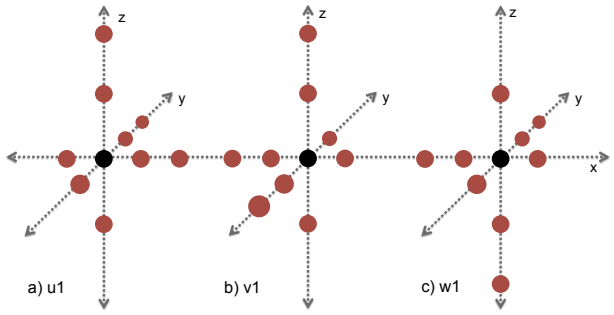


Figure 7.3: Stencil shapes of the velocity components used in the stress kernel. The kernel uses a subset of asymmetric 13-point stencil, coupling 12 points from u_1, v_1 and w_1 with central point accessed 3 times.

After computing the velocity, we update the stress components in another triply-nested loop using all three velocity components. We refer to this loop as *stress*. The stress kernel also employs the asymmetric 13-point stencil, shown in Fig. 7.3. The kernel refers to each velocity component 12 times with central points accessed 3 times. The pseudo-code in Table 7.2 shows the *xy* stress component computation, which is similar to that of the *xz* and *zy* components. The calculations of xx, yy and zz are a bit more involved, as they compute additional intermediate quantities. In the pseudo-code, as an example, we show only the *xx* component since *yy* and *zz* are similar to *xx*.

Table 7.3 lists the flop and memory access characteristics for the velocity and stress loops in the application. As presented in Chapter 6, the single precision flops:word ratios of the devices used in this thesis are 24.4 for the C1060 and 28.6 for the C2050. However, the velocity and stress kernels in AWP-ODC have flops:word ratios of 3.8 and 3.6 respectively. Thus, AWP-ODC is highly memory bandwidth bound and its performance depends on the device memory bandwidth rather than the floating point capacity of the devices.

7.1.4 Mint Implementation

The Mint implementation of the simulation required only modest programming effort. We added only 5 Mint `for` directives to annotate the two most time consuming loops and the three boundary condition loops in Table 7.2. In addition, we surrounded the time step loop with the parallel region pragma. Lastly, we added a data transfer pragma for each array prior to the start of the parallel region and retrieved the velocity and stress values from the GPU at

Table 7.2: Pseudo-code for the main loop, which contains the two most time-consuming loops, velocity and stress. c_1, c_2 and dt are scalar constants.

```
Main Loop:
Do T = time_step 0 to time_step N
      Compute velocities (u1, v1, w1) based on stresses (xx, yy ,zz, xy, xz, yz)
      Update boundary values of velocities based on (u1, v1)
      Update boundary values of velocity (w1) based on (u1, w1)
      Compute stresses (xx, yy ,zz, xy, xz, yz) based on velocities (u1, v1, w1)
      Update boundary values of stresses (xx, yy ,zz, xy, xz, yz)
End Do

Velocity: Compute velocity u1 (v1 and w1 are not shown)
foreach(i,j,k)
  u1[i,j,k] += d_1[i,j,k]*(c1*(xx[i,j,k] - xx[i-1,j,k]) + c2*(xx[i+1,j,k] -xx[i-2,j,k])
                          + c1*(xy[i,j,k] - xy[i,j-1,k]) + c2*(xy[i,j+1,k] - xy[i,j-2,k])
                          + c1*(xz[i,j,k] - xz[i,j,k-1]) + c2*(xz[i,j,k+1] - xz[i,j,k-2]))

Stress: Compute xx stress component (yy and zz are not shown)
foreach(i,j,k)
   vxx = c1*(u1[i+1,j,k] - u1[i,j,k]) + c2*(u1[i+2,j,k] - u1[i-1 ,j ,k])
   vyy = c1*(v1[i,j,k] - v1[i,j-1,k]) + c2*(v1[i ,j+1,k] - v1[i,j-2,k])
   vzz = c1*(w1[i,j,k] - w1[i,j,k-1]) + c2*(w1[i ,j,k+1] - w1[i,j,k-2])
   tmp = xl[i,j,k] * (vxx + vyy + vzz)
   a1 = qpa[i,j,k] * (vxx + vyy + vzz)
   xx[i,j,k] = xx[i,j,k] + dth * (tmp - xm[i,j,k] * (vyy + vzz)) + DT * r1[i,j,k]
   r1[i,j,k] = (vx2[i,j,k] * r1[i,j,k] - h[i,j,k] * (vyy+vzz) + a1) * vx1[i,j,k]
   xx[i,j,k] = (xx[i,j,k] + DT * r1[i,j,k] ) * dcrj[i,j,k]

Stress: Compute xy stress component (xy and yz are not shown)
foreach(i,j,k)
   vxy = c1*( u1[i,j+1,k] - u1[i,j,k]) + c2*(u1[i,j+2,k] - u1[i,j-1,k])
   vyx = c1*( v1[i,j,k] - v1[i-1,j,k]) + c2*(v1[i+1,j,k] - v1[i-2,j,k])
   xy[i,j,k] = ( xy[i,j,k] + xmu1[i,j,k] * (vxy + vyx) + vx1 * r4[i,j,k]) * dcrj[i,j,k]
   r4[i,j,k] = ( vx2[i,j,k] * r4[i,j,k] + h1[i,j,k] * (vxy+vyx)) * vx1[i,j,k]
   xy[i,j,k] = ( xy[i,j,k] + DT * r4[i,j,k]) * dcrj[i,j,k]
```

Table 7.3: Number of memory accesses, flops per element and flops:word ratio of the AWP-ODC kernels.

	Reads	Writes	Flops	Flops:Word
Velocity	13	3	60	3.8
Stress	28	12	142	3.6
Total	41	15	202	3.6

the end of the simulation. These pragmas count for a total of 40 copy directives (31 input and 9 output arrays). Compared with the extensive changes needed in the hand coded version–manage storage, handle data motion between host and device, map loop indices to thread indices, and outline CUDA kernels–Mint required only modest programming effort.

Only a portion of the AWP-ODC requires processing by Mint. The code that is not processed by Mint consists of 869 lines of code performing initialization and IO. The most time consuming part of the AWP-ODC, which is the input code to the Mint translator, is 185 lines of C code including 46 lines of Mint code (40 copy, 5 for-loop and 1 parallel region directives). The best performing variants for our two testbed devices are different. The Mint-generated code for the C2050 is 1185 lines and for the C1060 it is 1211. The device code of the C2050's best performer uses registers and L1 cache and counts for 384 lines of CUDA code. The C1060's best performer uses the shared memory optimization, which requires handling ghost cells, resulting in considerably more device code (434 lines). The rest of the Mint-generated code (777 lines) constitutes the host code required for kernel configuration, kernel launch, error check, and data transfer. The auto-generated code for data motion is lengthy, nearly 600 lines. This is because Mint doesn't generate simple calls to *cudaMemcpy* and *cudaMalloc*. It employs the *cudaPitchedPtr* data type to align arrays in device memory and *cudaMemcpy3D* for data transfer, which uses other CUDA specific data structures, *cudaExtent* and *cudaMemcpy3DParms*, as parameters. Mint also generates code to check whether each CUDA library call was successful. Table 7.4 summarizes the lines of code added and generated for CUDA acceleration of the simulation.

CUDA limits the kernel argument size to 256 bytes. This becomes an issue for an application using more than 30 arrays and referring to several scalar variables. Mint overcomes this restriction by packing arguments into a C-struct, passing as an argument to the kernel, and unpacking the struct inside the kernel. However, the user doesn't have to worry about such

Table 7.4: Summarizes the lines of code annotated and generated for the AWP-ODC simulation.

	Lines of Code
Original Code	185
Total Directives	46
Parallel directives	1
For-loop directives	5
Copy directives	40
Tesla C1060	
Generated Code	1211
Device Code	434
Host Code	777
Tesla C2050	
Generated Code	1185
Device Code	384
Host Code	801

programming details. Please refer to Chapter 4.2.9 for details.

7.1.5 Performance Results

We next evaluate various performance programming strategies realized by experimenting with Mint pragmas and translator optimization options. All results are for a simulation volume of 192^3 running for 800 timesteps. All arithmetic was performed in single precision. Table 7.5 shows the results for AWP-ODC Earthquake simulation on 3 platforms for the MPI implementation, hand-coded CUDA and Mint-generated CUDA variants (*Mint-baseline and Mint-opt*). The MPI version was previously implemented in Fortran by others and taken from the work [COJ+10] nominated for the Gordon Bell Prize in 2010.

As with the other applications, we refer to *baseline* as the plain Mint-generated code without any tuning and compiler optimization. Default values were used for the Mint pragma clauses: *nest(all)*, *tile(16x16x1)* and *chunksize(1,1,1)*. The baseline version does not utilize on-chip memory and naively makes all memory accesses through global memory. The *Mint-opt* variant was obtained without modifying any executable input source code. Rather, the user can try different performance enhancements via pragmas and command line compiler options. The *Mint-opt* refers to the variant employing an optimal mix of for-loop clauses: *tile* and *chunksize*

Table 7.5: Comparing performance of AWP-ODC on the two devices and a cluster of Nehalem processors (Triton). The Mint-generated code running on a single Tesla C2050 exceeds the performance of the MPI implementation running on 32 cores. Hand-CUDA refers to the hand coded (and optimized) CUDA version.

Platform	MPI Processes	Time(sec)/iteration	Gflop/s
Triton Intel E5530 (MPI)	1	0.4538	3.2
	2	0.2251	6.4
	4	0.1315	10.9
	8	0.0869	16.5
	16	0.0456	31.3
	32	0.0256	55.8
Tesla C1060 (200 series)	Mint-baseline	0.1651	8.7
	Mint-opt	0.0687	20.8
	Hand-CUDA	0.0540	26.5
Tesla C2050 (Fermi)	Mint-baseline	0.0597	23.9
	Mint-opt	0.0228	62.6
	Hand-CUDA	0.0189	75.8

and compiler options: *register, shared, preferL1*.

Table 7.5 reveals that, on the C1060, the performance of *Mint-baseline* is equivalent to that of the MPI variant running on between 2 and 4 Nehalem cores of the Triton cluster. The *Mint-opt* variant realizes performance equivalent to beween 8 and 16 Triton cores. On the C2050, *baseline* performance is equivalent to that of between 8 and 16 cores, and *Mint-opt* (62.6 Gflop/s) runs slightly faster than 32 Triton cores (55.8 Gflop/s). We attribute the performance difference between the C2050 and the C1060 to the memory bandwidth. Compared with the C1060, the C2050 sustains 65% more read bandwidth and 75% more write bandwidth to the device memory. Mint optimizations (shown as *Mint-opt*) improve the performance by a factor of 2.4 and 2.6 on the C1060 and C2050, respectively. The best-quality code produced by Mint achieves 82.6% of the hand-coded implementation (shown as *Hand-CUDA* in Table 7.5) on the C2050 and 78.6% on the C1060.

Next, we take a close look at performance programming and analyze the effect of the Mint clauses and compiler options on application performance. Moreover, we evaluate the hand-coded version and compare it with the Mint variants.

7.1.6 Performance Impact of Nest and Tile Clauses

Using the *nest* clause, Mint enables the user to control the depth of data parallelism. In Chapter 6, we have previously demonstrated the superiority of nested parallelism over the single level parallelism in 3D stencil methods. In short, parallelizing all levels of a loop nest improves inter-thread locality (coalesced accesses) and also leads to vastly improved occupancy. We therefore used the `nest(all)` clause on all the loops to parallelize AWP-ODC. This would correspond to using *nest(3)* for the triply nested loops in Table 7.2, and *nest(2)* for the doubly nested loops that compute boundary conditions.

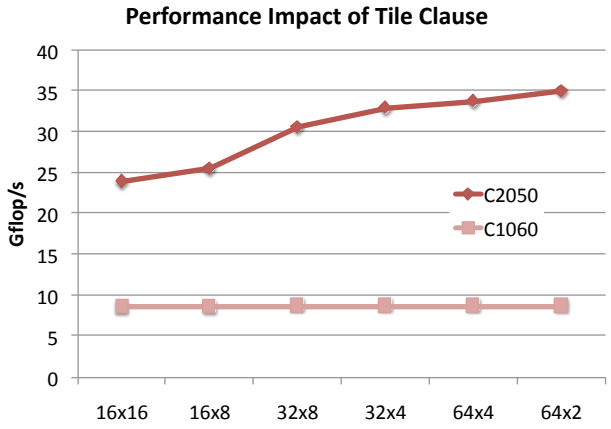


Figure 7.4: Experimenting different values for the tile clause. On the Tesla C2050, the configuration of `nest(all)` and `tile(64,2,1)` leads to the best performance.

We found that the optimal tile size for AWP-ODC depended on the hardware (Fig.7.4). The hardware places an effective lower bound on the x-dimension (fastest varying dimension) of the thread block geometry. To ensure coalesced memory access, a thread block requires at least 16 threads in the x-dimension. We found that we were able to improve performance by extending the x-dimension of a thread block, rendering an elongated thread block.

The best tile size on the C1060 was 32×8, although the sensitivity of performance to tile size was low[1]. Since the tile size (32×8) doesn't provide a significant performance benefit, we favor the second best but smaller tile size (32×4). A smaller tile size can compensate for the register requirements when we apply the Mint compiler options (discussed next), which increase the demand for registers. On the C2050, we found that a 64×2 tile was optimal, raising the performance of the Mint-baseline version from 24.0 to 35.0 Gflop/s. The poor performance for

[1]Through an abuse of notation, we drop the trailing 1 to indicate a 2-dimensional geometry.

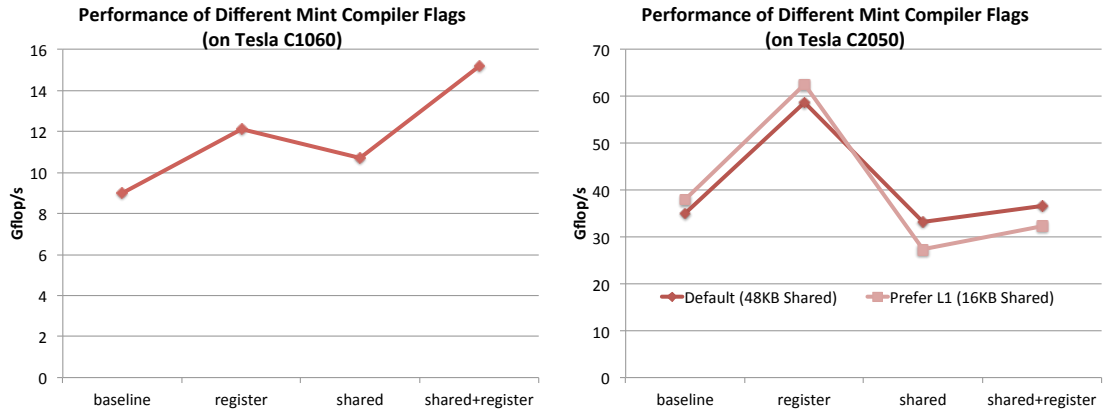


Figure 7.5: On the C1060 (left), the best performance is achieved when both the shared memory and register flags are used. The results are with the `nest(all), tile(32,4,1)` and `chunksize(1,1,1)` clause configurations. The shared flag is set to 8. On the C2050 (right), the best performance is achieved when a larger L1 cache and registers are used. The results are with the `nest(all), tile(64,2,1)` and `chunksize(1,1,1)` configurations. The shared flag is set to 8.

the default tile size (16×16) stems from the fact that the C2050 has a L1 cache line size of 128 bytes. For an application using single precision arithmetic, each thread reads 4 bytes of data from a cache line. This translates into 32 threads in a warp reading from the same cache line. A smaller tile size would underutilize the load units. A tile size that is a multiple of 32 in the x-dimension therefore improve performance.

Next, we discuss the impact of compiler options for the fixed tile clauses.

7.1.7 Performance Tuning with Compiler Options

As mentioned previously, Mint provides three optimization flags to tune performance: *register, shared* and *preferL1*. The left side of Fig. 7.5 shows the performance impact of these flags on the Tesla C1060. Since there is no L1 cache on the C1060, the preferL1 option is not used. The *register* option improves the baseline performance by 35% as it places the frequently accessed data, mainly central stencil points, into registers. Although a compiler may be able to detect such reuse in simple codes, complex loops with a large number of statements referencing several arrays are problematic. Mint, being a domain-specific translator, expects such reuse in the stencil kernels and is able to capture this information to better utilize the register file. We can observe direct evidence of this behavior when we compile Mint generated CUDA code with

nvcc. The baseline code uses unnecessarily more registers than when the Mint register optimizer is enabled. This shows that *nvcc* cannot detect the data reuse in the code.

The best performance on the C1060 is achieved when both register and shared memory optimizations are turned on. Shared memory buffers global memory accesses and acts as an explicitly managed cache. However, effective shared memory optimization is contingent on register optimization. Shared memory optimization alone leads to lower performance than when both shared memory and registers are used. This is because when registers augment shared memory, some of the operands of an instruction can reside in registers rather than in shared memory. An instruction executes more quickly (up to 50%) if its operands are in registers than they are in shared memory [VD08].

The right side of Fig. 7.5 presents the results for the C2050. The performance benefit of shared memory disappears on the C2050 architecture. Instead, prefering L1 cache combined with the register optimizer is more advantageous. This is primarily true because shared memory usage also increases the demand for registers which can be counterproductive. This is an artifact of the *nvcc* compiler. We observed that on the 2.0 capable devices the Nvidia compiler uses more registers if the kernel employs shared memory. Excessive register and shared memory pressure can lower device occupancy to a level where there are not sufficient threads to hide global memory latency. Since the number of registers per core on Fermi devices drops to half compared to the 200-series, registers are more precious resource. On the other hand, cache does not increase register pressure in the same way as shared memory does. In contrast, occupancy should not be maximized to the exclusion of properly managed on-chip locality on the C1060 because there is no L1 cache and the memory bandwidth is lower than on the C2050.

7.1.8 Shared Memory Option

We have experimented with different ways of mapping data onto shared memory on both devices even though shared memory was not the winning optimization on the C2050. Fig.7.6 plots the performance with different limits for the shared flag (e.g., *shared=1*). The limit sets the upper limit for the available shared memory slots to the kernel. The increase in the shared memory usage helps improve performance on the C1060. But as explained previously, shared memory degrades performance on the C2050.

Fig. 7.7 depicts the selected variables for the shared memory slots for both kernels. The selection algorithm, presented in Section 5.3.5, creates two lists for each kernel: 1-plane list and 3-planes list. Since the top and bottom points can be referenced from registers, there is no

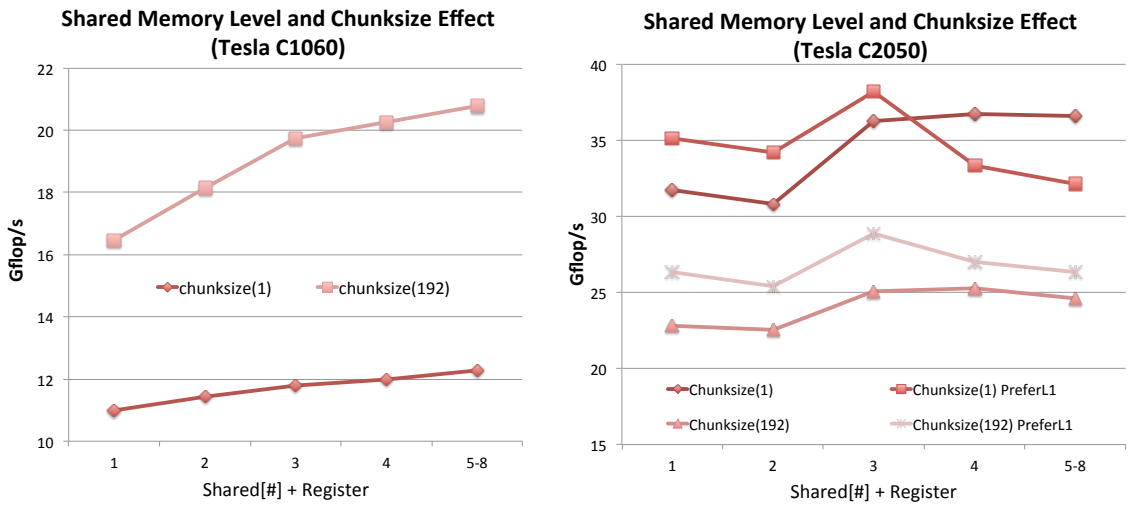


Figure 7.6: On the C1060 (left), chunksize and shared memory optimizations improve the performance but on the C2050 (right), these optimizations are counterproductive.

sharing along the z-dimension for both kernels. Thus, their 3-planes list is empty. To be eligible for being in the 1-plane list, there should be at least one off-center reference in the xy-plane.

Setting *shared=1* restricts shared memory to a single slot per kernel. Based on the number of references and stencil pattern, Mint picks the best candidate variable for each kernel. For example, in the *velocity* kernel, 5 stress variables are candidates for shared memory because they have shareable references in their xy-plane. The pattern is a 13-point stencil (see Fig 7.2) but asymmetric; only a subset of 13 points is referenced and the subset for each variable is different. Mint chooses to place the xy grid in shared memory, since 8 references are made to this array, all of which lie in an xy-plane as shown in Fig. 7.2. Similarly, yz and xz are referenced 8 times but their memory locations expand in the z-dimension (slowest varying dimension) and these references are not shared by a 2D thread block, only 5 of them are shared. Mint picks a variable that uses the fewest planes but saves the most trips to global memory. The 3 other stress variables are not strong candidates because they are accessed only 4 times. If the register optimizer is turned on together with the shared memory optimizer, then Mint places the central stencil into registers. For example, 2 out of the 8 references to the xy grid access the central point, so Mint assigns these to registers.

Shared=2 increases the variable count to 2 per kernel and *shared=3* to 3. In the stress kernel, there are 3 candidate arrays (v_1, u_1, w_1) for shared memory and each is referenced 12 times. Out of these 12 references, 8 are made to the xy-plane, as shown in Fig. 7.3. Mint loads these grids into shared memory and assigns one plane to each because doing so saves 8 references

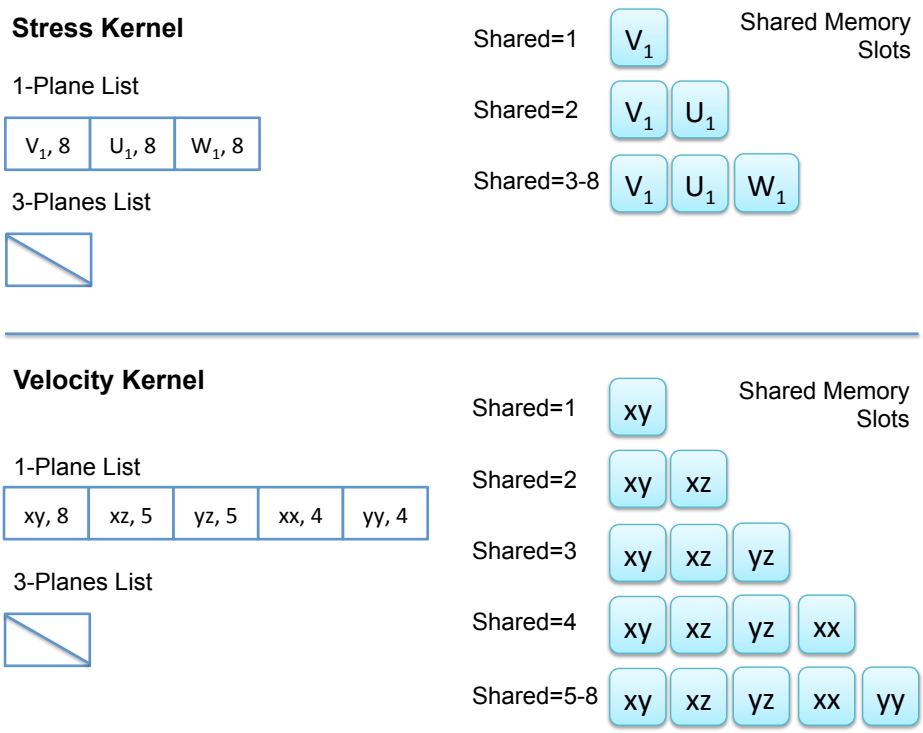


Figure 7.7: Result of selection algorithm for shared memory slots

for each variable. However, these variables refer to the central point and 3 other points in the z-dimension with offsets of -1, $+1$, and $+2$ (or -2). There is no sharing of points with threads residing in the same thread block along the z-dimension. Therefore, Mint does not assign more than one plane to each of these grids. These references can be improved with registers instead.

For the slot limit *shared=4* and *5,* Mint increases only the *velocity* kernel's variable count to 4 and 5 in shared memory because the other kernel has only 3 candidates for the 4 shared memory slots and assigning more planes to a variable did not save any further global memory references. In other words, *shared=3-8* has the same effect on the *stress* kernel. We can clearly see this in performance results of the C1060. After *shared=3*, the performance improvement is modest because only the *velocity* kernel takes advantage of the further increased shared memory usage. Mint generates the same CUDA code for the values from 5 through 8 for AWP-ODC since there are more shared memory slots than good candidates.

7.1.9 Chunksize Clause

When the chunksize is set to 1 (the default), each CUDA thread updates a single iteration in the z-dimension. However, in stencil computations there is a performance benefit to

aggregating loop iterations so that each CUDA thread computes more than one point. We have applied chunking along the slowest varying dimension and set it to cover an entire z-column of 192 points. If chunking is combined with shared memory and register options, Mint can reuse the data along the z-column. Fig 7.8 illustrates the effect of chunking on the *stress* kernel. Mint keeps the xy-plane in the shared memory, but keeps the center (0,0,0), top (0,0,+1) and bottom (0,0,-1) points in the register file. Since the same thread computes all the points in a single z-column, Mint moves the content of the registers from the bottom up before updating the bottom with a new load from global memory.

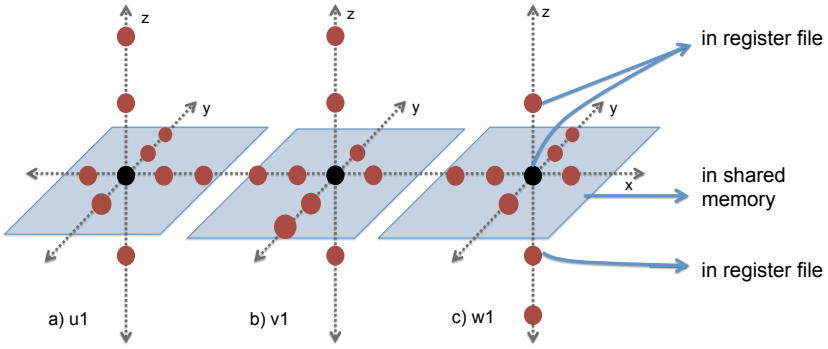


Figure 7.8: shows where the data is kept when both register and shared memory optimizers are turned on and chunking is used.

Chunking increases performance by 70% on the C1060 (see Fig. 7.6). The best performance (20.8 Gflop/s) is achieved with the following configuration: *register, shared=5*, *tile(32,4)* and *chunksize=192*. A thread block size of 32×8 would not allow the chunksize optimization, resulting in 0 thread blocks because the kernel would exceed the hardware register limit on the C1060. The *velocity* kernel employs 50 registers and 5800 bytes of shared memory, running two thread blocks per multiprocessor (25% occupancy). The *stress* kernel uses 91 registers and 3504 bytes of shared memory, resulting in 13% multiprocessor occupancy, running a single thread block. Obtaining high performance with low occupancy on the 200-series supports our previous claim that the properly managed on-chip memory can overcome the performance loss due to the low device occupancy.

As with the shared memory optimization, chunksize optimization increases register pressure, which can be counterproductive. On the C2050, chunking has a devastating effect and reduces performance by about one-third. This is because the Fermi architecture limits the number of registers per thread to 63, which is 127 on the 200-series. The Nvidia C compiler spills registers to local memory when this limit is exceeded. The latency to local memory is as high as

to global memory. Unfortunately, both the shared memory and chunksize variants cause register spilling on the C2050. For example, the chunksize variant of the *stress* kernels spills 324 bytes of stores and 312 bytes of loads per thread. According to the *Nvidia Fermi tuning guide* [Nvi10b], favoring a larger L1 may cache spilled registers. However, this may lead to contention with other data in the cache.

7.1.10 Analysis of Individual Kernels

The Mint for-loop clauses are local to the loop and the programmer can set different values for the clauses for different kernels. However, the Mint compiler flags are global and applied to the entire program. A compiler configuration may improve the performance of one kernel but at the same time degrade another. In order to see the performance impact of various tile clauses and compiler options on individual kernels in AWP-ODC, we closely examined the runing time of each kernel. Fig. 7.9 shows the analysis of running time on the C1060 for the two most time-consuming loops[2]: *velocity* and *stress*. Note that the trend for running times is the opposite of Gflop/s rates.

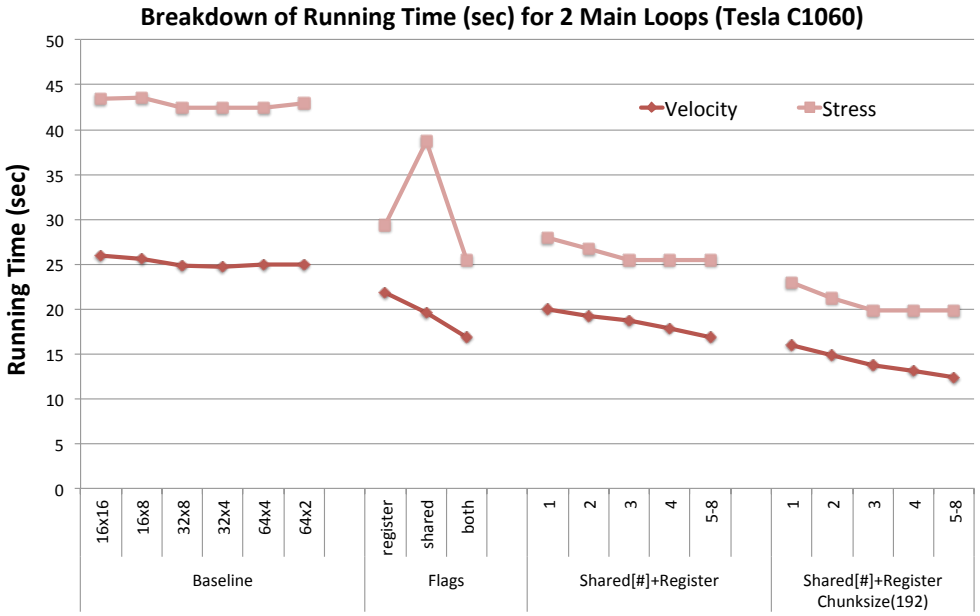


Figure 7.9: Running time (sec) for the most time-consuming kernels in AWP-ODC for 400 iterations on the Tesla C1060. Lower is better. The compiler flag "both" indicates shared+register. Lower is better.

[2]The contribution of the boundary condition loops to the running time is negligable.

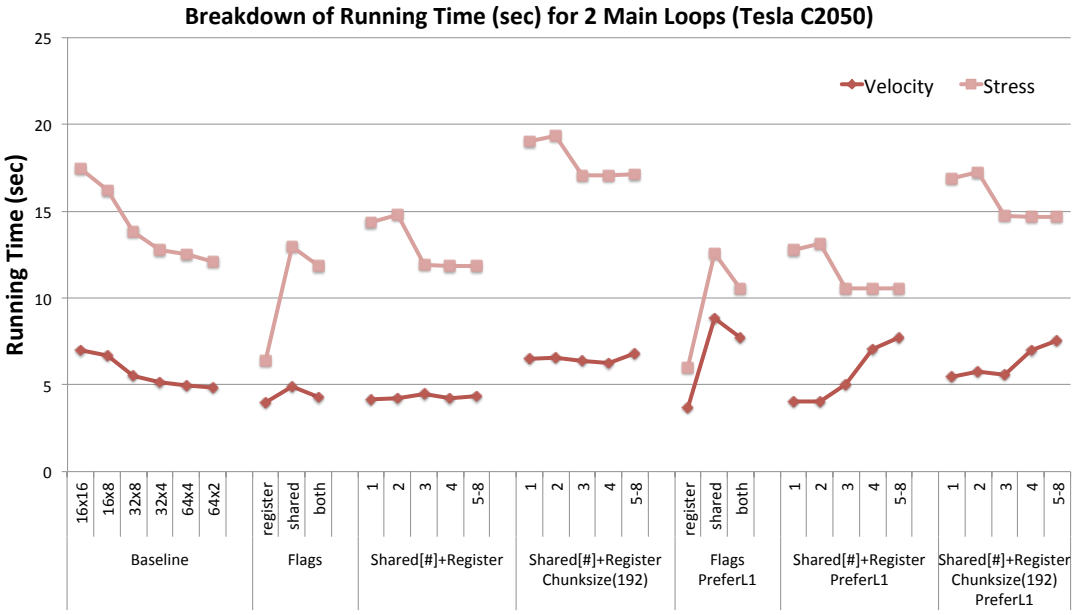


Figure 7.10: Running time (sec) for the most time-consuming kernels in AWP-ODC for 400 iterations on the 2050. Lower is better. The compiler flag "both" indicates shared+register. Lower is better.

On the C1060, both kernels follow a similar trend as we change the configurations except for the *shared* flag. The tile size does not have notable impact on the performance. While the *stress* kernel cannot benefit from shared memory without the assistance of registers, shared memory still improves the performance of *velocity*. This is because the *stress* kernel has a larger loop body and the shared memory optimizer requires twice the number of registers as required by the *velocity* kernel. Increasing shared memory usage combined with register optimization is helpful though since Mint uses registers effectively. As we discussed previously, the choice of *shared=3-8* generates the same code for the *stress* kernel. Thus performance is steady after *shared=3*. The best performance for the individual kernels is achieved when the chunksize, shared and register optimizations are enabled, which is consistent with the best overall performance for the entire application.

On the C2050 (Fig 7.10), both kernels have similar trends for different tile sizes. Hence the local optimal tile is the same as the global, which is 64×2. The kernels exhibit differences in how they react to the shared memory limit. Increasing shared memory usage while optimizing for registers appears to improve the performance of the *stress* kernel. However, the best performance is achieved when the shared memory is not used at all. This shows that the performance

loss due to the register spilling is not compensated for the performance gain due to the reduction in global memory references through shared memory. Without the preferL1 option, the change in the limit does not affect the performance of the *velocity* kernel. With the perferL1 option, it lowers the performance of the *velocity*. Even though the *stress* kernel takes more than twice the time of the *velocity*, the performance loss due to the shared memory usage in the *velocity* kernel is significant enough to influence the aggregated performance. For both kernels, the best performance is achieved when the register and preferL1 options are turned on. Neither of the kernels benefit from chunksize optimization because of the register limit on the C2050.

On both devices, on-chip memory optimizations improve the *stress* kernel performance slightly more than *velocity*. Without optimizations, the *stress* kernel takes 63% and 72% of the total execution time on the Tesla C1060 and C2050 respectively. After optimization, the kernel takes 61% (C1060) and 62% (C2050) of the total time. The main reason is that in the *stress* kernel, stencil-like global memory references are concentrated on few arrays ($u1, v1$ and $w1$) each with high access frequencies. On the other hand, the velocity kernel has 6 stencil arrays each with lower access frequencies. As a result, the *stress* kernel is more amenable to the on-chip memory optimizations because it saves more references to global memory. On both devices, the kernel-level configuration of the for-loop clauses and compiler options that gives the best performance is the same as the program-level configuration.

7.1.11 Hand-coded vs Mint

Mint achieved 82.6% and 78.6% of the hand-written and optimized CUDA code on the C2050 and C1060, respectively. Considering that the hand-CUDA implementation took a long time and lots of programming effort to reach its current performance level, we think this performance gap is reasonable in light of the simplified programming model enjoyed by Mint's pragma-based model. When the programmer wants to have the highest performance possible, Mint can constitute a lower bound for a hand-written code. Mint can motivate a developer to set a more ambitious performance goal although it would take a lot of time to close the last 20% gap.

Similar to the Mint variants, the hand-optimized CUDA variants are different for each device. The hand-optimized version for the C2050 primarily utilizes registers and does not use shared memory. This is consistent with the best-quality code generated by Mint. On the 200-series, the shared memory variant with chunking was the winning implementation as with Mint. However, unlike the Mint variant, the hand-coded variant uses texture memory and constant

memory on both devices. Mint currently does not support either of these on-chip stores. To better understand the effect, we hand-modified the best-quality kernel generated by Mint for the C2050 to use texture memory. However, we observed only a modest performance improvement (3.2%).

The performance difference between Mint and the hand-optimized code mainly stems from the fact that the hand-written chunksize optimization uses far fewer registers than Mint. As we discussed in the previous section, this optimization requires a large number of registers. However, Mint generates code that uses more registers than the hand-optimized code that implements the same optimization strategy. A programmer can do a better job in allocating registers than a compiler. He has the freedom to manually reorder the instructions and reuse the registers that are no longer needed. Mint relies on `nvcc` for reordering and register allocation.

Another way in which the implementations differ is in how they treat padding, which helps ensure that memory accesses are aligned. Mint always uses *cudaMalloc3D* along with *cudaPitchedPtr* to pad the storage allocation. As a result, it aligns memory to the start of the mesh array, which includes the ghost cells. On the other hand, the hand-optimized CUDA variant simply pads zero at the boundaries so that memory is aligned to the inner region of the input arrays, where the solution is updated. Without padding, the performance of hand-optimized CUDA dropped to 66.0 Gflop/s from 75.8 Gflop/s.

7.1.12 Summary

We have demonstrated an effective approach to porting a production code of anelastic wave propagation using Mint. Mint generated code realized about 80% of the performance of hand-coded CUDA. Compared with writing CUDA by hand, Mint provide the flexibility to explore different performance variants of the same program through annotations and compiler options. For this study, we generated nearly 50 variants of the AWP-ODC simulation with Mint. In addition to the variants as a result of different compiler options, any changes made into the algorithm by the application developers resulted in minor to major changes in the optimized code. For example, we experimented splitting the stress subroutine into two subroutines, which doubles the optimization space. It is very time-consuming for a programmer to implement each of these variants by hand. In particular, implementing shared memory variants is very cumbersome due to the management of ghost-cells and shared memory slots. A Mint programmer can generate various shared memory variants by simply setting the shared flag (*shared=[1-8]*).

Shared memory usage increases the use of registers which makes this optimization coun-

terproductive in some cases especially on the C2050. Since the hardware limit on the number of registers per thread is half that of on the 400-series (63 vs 127), there is not much room for improvement with shared memory on the C2050 device. If the hardware limit is hit, the Nvidia C compiler spills registers to local memory which is detrimental to performance because of the access cost.

7.2 Harris Interest Point Detection Algorithm

7.2.1 Background

Feature detection plays a very important role in many computer vision algorithms. It finds features in an image such as corners or edges that would help the viewer understand the image. In order to apply advanced algorithms, such as object recognition or motion detection, it is essential to have a good feature detection algorithm. One of the widely used feature selection algorithms for 2D images is the Harris interest point detection algorithm [HS88], which produces a score for each pixel in the image. High scores indicate "interest" points. Kim et. al [KUSB12] extended the algorithm to 3D volume datasets. In volume visualization, such algorithms can be applied to transfer function manipulation [LBS+01], or to control colors and opacities of objects in the data [PLS+00, KKH02, KSC+10].

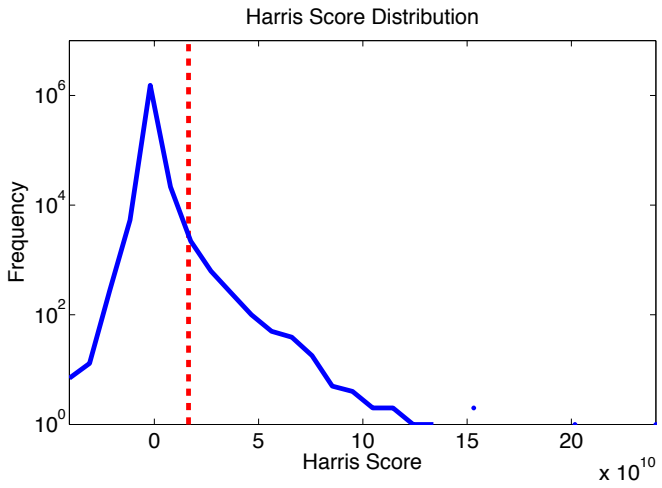
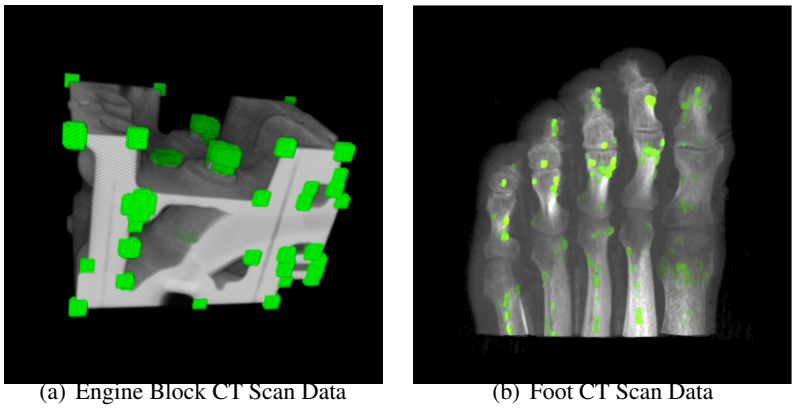


Figure 7.11: The Harris score distribution of a volume dataset. The solid line shows the histogram of the Harris score. Large positive values are considered interest points or corner points, near zero points are flat areas, and negative values indicate edges. The dotted line shows the threshold.

7.2.2 Interest Point Detection Algorithm

The Harris interest point detection algorithm [HS88] extracts a set of features from an image by scoring the importance level of each pixel in the image. Large positive score values correspond to interest points. A pixel is selected as a point of interest if there is a large change

both in the X- and Y- directions. For example, corners of an object or high intensity points get a large positive score. On the contrary, the algorithm assigns a score close to zero to homogenous regions and large negative scores to edges. Fig.7.11 shows the Harris score distribution for a volume dataset. All the points above a predetermined threshold are considered to be interest points.



(a) Engine Block CT Scan Data (b) Foot CT Scan Data

Figure 7.12: The Harris corner detection algorithm applied to the Engine Block CT Scan from General Electric, USA and the Foot CT Scan from Philips Research, Hamburg, Germany. The algorithm identified the corners of the engine block and around the joints in the foot image as interest points, shown with green squares.

Fig. 7.12 shows the interest points detected by the 3D Harris algorithm for two images. Fig. 7.12(a) shows an engine block with green squares highlighting the features identified by the algorithm. All the corners and the two cylinders on the top were detected as features. Fig. 7.12(b) shows a foot CT scan, which does not have distinct corners. However, the algorithm successfully identifies the tips and joints of the toes as features.

The basic idea of computing the Harris score is to measure the change E around a voxel (x,y,z) in a 3D object:

$$E(x,y,z) = \Sigma_{r,s,t} g(r,s,t)|I(x+r,y+s,z+t) - I(x,y,z)|^2 \tag{7.13}$$

where $I(x,y,z)$ is the opacity value at image coordinate (x,y,z) and $g(r,s,t)$ is defined as a Gaussian weight around (x,y,z). The Taylor expansion for $I(x+r,y+s,z+t)$ is approximated as follows:

$$I(x+r,y+s,z+t) \approx I(x,y,z) + xI_x + yI_y + zI_z + O(x^2,y^2,z^2) \tag{7.14}$$

Then E can be written in matrix form:

$$
\begin{aligned}
& E(x,y,z) \\
& = \Sigma_{r,s,t} g(r,s,t) |I(x+r,y+s,z+t) - I(x,y,z)|^2 \\
& = \Sigma_{r,s,t} g(r,s,t) |xI_x + yI_y + zI_z|^2 \\
& = \begin{bmatrix} x & y & z \end{bmatrix} \begin{bmatrix} g \otimes I_x^2 & g \otimes I_x I_y & g \otimes I_x I_z \\ g \otimes I_x I_y & g \otimes I_y^2 & g \otimes I_y I_z \\ g \otimes I_x I_z & g \otimes I_y I_z & g \otimes I_z^2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \\
& = \begin{bmatrix} x & y & z \end{bmatrix} M(x,y,z) \begin{bmatrix} x & y & z \end{bmatrix}^T
\end{aligned} \tag{7.15}
$$

This equation says that the matrix M in Eq. 7.15 defines the changes at point (x,y,z), and the three eigenvalues of this matrix characterize the changes on each principal axis. If the change is large on all three axes, i.e., all three eigenvalues are large, the point is classified as a corner. On the other hand, for the points with a small change, i.e., homogeneous areas, M has small eigenvalues. If M has one or two large eigenvalues, it indicates that the point is on an edge. The eigenvalues of M can be computed by singular value decomposition, which is computationally expensive. Alternatively, Harris and Stephens [HS88] proposed a response function as follows:

$$R = det(M) - kTrace(M) \tag{7.16}$$

where k is an empirical constant, and R is the “Harris score" of a voxel. If the eigenvalues are significantly large, the determinant of M is large, thus R is positive. If only one or two eigenvalues are significant, R becomes negative as $Trace(M)$ is bigger.

In our implementation, we set the the width of the Gaussian convolution as 5 in all axes; the convolution is a weighted sum of a $5 \times 5 \times 5$ patch around a point (x,y,z). The variance of the Gaussian weight is set to 1.5, and the sensitivity constant k is set to 0.004.

7.2.3 The Stencil Structure and Storage Requirements

The algorithm takes the opacity data and produces a Harris score for each voxel. The kernel requires many memory accesses and arithmetic operations per voxel due to the convolution sum. The convolution is a weighted sum of a patch around a point (x,y,z). The patch size w affects the accuracy of the feature selection. Larger windows result in a more accurate solution but take longer time to run. The algorithm computes the gradient for x-,y- and z- directions for each voxel in a patch. There are 6 references to the voxel array when computing gradients (2 per

axis). This translates into $w^3 * 6$ references per voxel. Fig.7.13 illustrates the memory references for a 2D case, where the window size is set to 5.

A convolution operation requires 21 floating point operations (9 add, 12 multiply). As a result, the kernel performs $21 * w^3 * 6$ operations for the convolution. Then it computes the Harris score by taking the determinant of matrix M, which adds 27 more floating point operations, resulting in $21 * w^3 * 6 + 27$ operations. For example, for a 5^3 patch, we would perform 125 memory references and 342 (6*21*125 + 27) floating point operations. However, convolution exhibits high reuse of data. The Mint translator exploits this property and greatly reduces the memory references (to be discussed shortly).

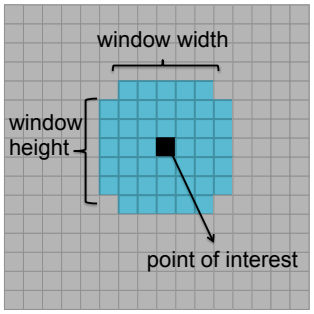


Figure 7.13: Coverage of a Gaussian convolution for a pixel in a 2D image.

7.2.4 Mint Implementation

The Mint program that implements the main loop of the 3D Harris appears in Table 7.6. The code computes the convolution and obtains a Harris score for each voxel. In a triple-nested loop from Line 7-9, we visit every voxel in a 3D volume datasets. For each voxel, we compute the Harris score in another triple-nested loop, which performs the Gaussian convolution. In this example, the window size of the Gaussian convolution is set to 5: we compute the convolution as the weighted sum of a $5\times5\times5$ patch around a voxel (x,y,z).

Line 1 copies the *data* voxel array from the host memory to device memory. The last two arguments of the `copy` directive indicate the dimensions of the image. Line 2 copies the weight vector to the device. Line 4 indicates the start of the accelerated region. The Mint `for` directive at line 6 enables the translator to parallelize the loop-nest on lines 7 through 41. The `nest(3)` clause specifies that the triple nested loop can run in parallel. The 3 inner loops starting at line 14 sweep the $5\times5\times5$ window. The translator subdivides the input grid into 3D tiles for

locality. The `chunksize` clause specifies the workload of a thread. In this program, a thread is assigned to 64 iterations in the outer loop. Lastly, Line 43 copies the Harris scores back to the host memory.

7.2.5 Index Expression Analysis

As we discussed in Chapter 5 the Mint optimizer analyzes the structure of the stencils and their neighbors because the shape of the stencil affects ghost cell loads (halos) and the amount of shared memory that is needed. The analyzer first determines the stencil coverage by examining the pattern of array subscripts with a small offset from the central point. That is, the index expressions are of the form $i \pm c$, where i is an index variable and c is a small constant. However, in the Harris algorithm, the array indices are relative to the Gaussian convolution loops (lines 14-16) in Table 7.6. The indices are not a direct offset to the main loops (lines 7-9). This makes it difficult for a compiler to analyze the nearest neighbor between the points. We implemented a pre-analysis step in Mint, discussed in Section 5.4, to allow the translator to handle such cases so that the stencil analyzer can effectively determine the stencil pattern appearing in the computation.

In the Harris algorithm, an index to the *data* array (e.g. Line 18 in Table 7.6) is a function of l, m and n which are functions of i, j, and k. In order to eliminate the l, m and n indices from the loop body, the translator rewrites the references to arrays in terms of i, j and k. First, it determines the unrolling factor and recursively unrolls the loops. The unrolling factor is the difference between the upper and lower bounds of the loop: the window size. In this example, it is 5 (line 14-16 in Table 7.6). After unrolling, the translator replaces the instances of l, m and n with i, j and k respectively. This process introduces expressions such as $i \pm c_1 \pm c_2$, that involve the index variable and a number of constants. To simplify the index expressions, we apply constant folding. This optimization converts, if possible, the index expressions, into the form $i \pm c$, where c is a small constant. After these transformations, the stencil analyzer can detect the stencil patterns appearing in the loop body.

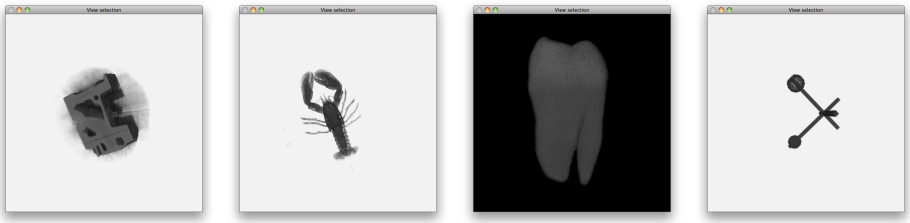
7.2.6 Volume Datasets

To collect performance results, we used four well-known volume datasets in volume rendering, also shown in Fig.7.14. All four datasets represent an image as a 3D uniform array of bytes. The pixel values range from 0 to 255, indicating the opacity of each point. The first dataset, *Engine*, is CT scan data from General Electric, USA and the second dataset, *Lobster*, is

Table 7.6: Main loop for the 3D Harris interest point detection algorithm.

```
#pragma mint copy(data, toDevice, width, height, depth)
#pragma mint copy(w, toDevice, 5, 5, 5)

#pragma mint parallel
{ // main loop
#pragma mint for nest(3) tile(16,16,64) chunksize(1,1,64)
  for (i = 3; i < depth - 3; ++i) {
    for (j = 3; j < height - 3; ++j) {
      for (k = 3; k < width - 3; ++k) {

        float Lx = 0.0f, Ly = 0.0f, Lz = 0.0f;
        float LxLy = 0.0f, LyLz = 0.0f, LzLx = 0.0f;

        for (l = i - 2; l <= i + 2; ++l) {
          for (m = j - 2; m <= j + 2; ++m) {
            for (n = k - 2; n <= k + 2; ++n) {
              // gradient in x direction
              float dx = (data[l][m][n+1] - data[l][m][n-1]);
              // gradient in y direction
              float dy = (data[l][m+1][n] - data[l][m-1][n]);
              // gradient in z direction
              float dz = (data[l+1][m][n] - data[l-1][m][n]);

              const float weight = w[l-i+2][m-j+2][n-k+2];
              // gaussian convolution sum
              Lx += weight * dx * dx;
              Ly += weight * dy * dy;
              Lz += weight * dz * dz;
              LxLy += weight * dx * dy;
              LyLz += weight * dy * dz;
              LzLx += weight * dz * dx;
            }
          }
        }
        // compute cornerness metric, Harris Scores
         harrisScores[i][j][k] = (Lx * Ly * Lz + LxLy * LyLz * LzLx + LxLy * LyLz * LzLx
                                  - LzLx * LzLx * Ly - LxLy * LxLy * Lz - LyLz * LyLz * Lx)
                       - sensitivity_factor * (Lx + Ly + Lz)* (Lx + Ly + Lz)* (Lx + Ly + Lz);
      }
    }
  }//end of nested-for
}//end of parallel region
#pragma mint copy(harrisScores, fromDevice, width, height, depth)
```

also a CT scan from the VolVis distribution of SUNY Stony Brook, NY, USA. The *Tooth* data has scanned with the GE Industrial Micro CT scanner. Finally, the *Cross* data is an artificial dataset created by Ove Sommer, in the Computer Graphics Group of the University of Erlangen, Germany. Table 7.7 shows the sizes of the volume datasets used in the experiment.



(a) Engine (b) Lobster (c) Tooth (d) Cross

Figure 7.14: Four well-known volume datasets in volume rendering

Table 7.7: Sizes of the volume datasets used in the experiment

Dataset	Size	Number of Voxels
Engine	$256 \times 256 \times 256$	16,777,216
Lobster	$301 \times 324 \times 56$	5,461,344
Tooth	$94 \times 103 \times 161$	1,558,802
Cross	$66 \times 66 \times 66$	287,496

7.2.7 Performance Results

We next demonstrate the effectiveness of Mint by comparing the performance of the Mint-generated CUDA and OpenMP implementations. The notation OpenMP(8) designates an OpenMP implementation running with 8 threads. OpenMP(1) indicates the single-threaded performance of the OpenMP code. We obtained the OpenMP results on the Triton cluster based on Nehalem processors. Table 7.8 lists the running times in seconds for 4 volume datasets with a $5 \times 5 \times 5$ convolution size.

We did not implement a hand-optimized CUDA version of the algorithm because Mint already enabled us to realize real time performance goals in 3D images. All the Mint-generated kernels run under 1 second, enabling interactive, i.e. real time, feature selection. On the C1060, the Mint translator delivers 5.9 to 9.8 times the performance of OpenMP(8). On the C2050

Table 7.8: Comparing the running time in seconds for different implementations of the Harris interest point detection algorithm, using four volume datasets with a 5×5×5 convolution window. The Tesla C2050 is configured as 48KB shared memory and 16KB L1 cache.

Dataset	Intel Xeon E5530		Tesla C1060		Tesla C2050	
	OpenMP(1)	OpenMP(8)	Mint-baseline	Mint-opt	Mint-baseline	Mint-opt
Engine	14.043	1.765	1.056	0.390	0.290	0.169
Lobster	4.578	1.299	0.346	0.132	0.095	0.057
Tooth	1.304	0.260	0.126	0.044	0.026	0.017
Cross	0.240	0.072	0.033	0.011	0.007	0.004

device, the speedup ranges from 10.5 to 22.8 over OpenMP(8). The speedup is higher for large datasets because we can effectively occupy all the stream processors on the GPU device and better hide the memory latency.

7.2.8 Performance Tuning with Compiler Options

The input data that stores the voxels is a good candidate for on-chip memory optimizations because there is a high degree of reuse. Next we discuss the impact of the Mint on-chip memory optimizer on performance.

Fig. 7.15 and Fig.7.16 show the cumulative performance improvements that result from applying optimizations for 5×5×5 convolution applied to 4 datasets on the C1060 and on the C2050, respectively. *Baseline* refers to the performance of the code generated by the Mint baseline translator without any optimizations applied. This variant resolves all array references through device memory. *Register* enables the *-register* flag, which uses registers to accommodate some of the global memory references. *Shared* enables the *-shared flag*, which uses shared memory to further improve the reuse of data by buffering memory accesses. Lastly, the *chunksize* clause indicates the performance when a chunking factor is set for the nested-loop in the input program. In this variant, each thread computes multiple elements in the outer loop as opposed to a single element as in previous variants. We set the chunking factor to 64 to obtain the results. In all Mint variants, loops are annotated with `nest(3)` to create multi-dimensional thread blocks.

As shown in Fig. 7.15 and Fig. 7.16, both register and shared memory optimizations substantially improve the performance over the baseline translation because they reduce the number of memory accesses to device memory. The average performance improvements of the register

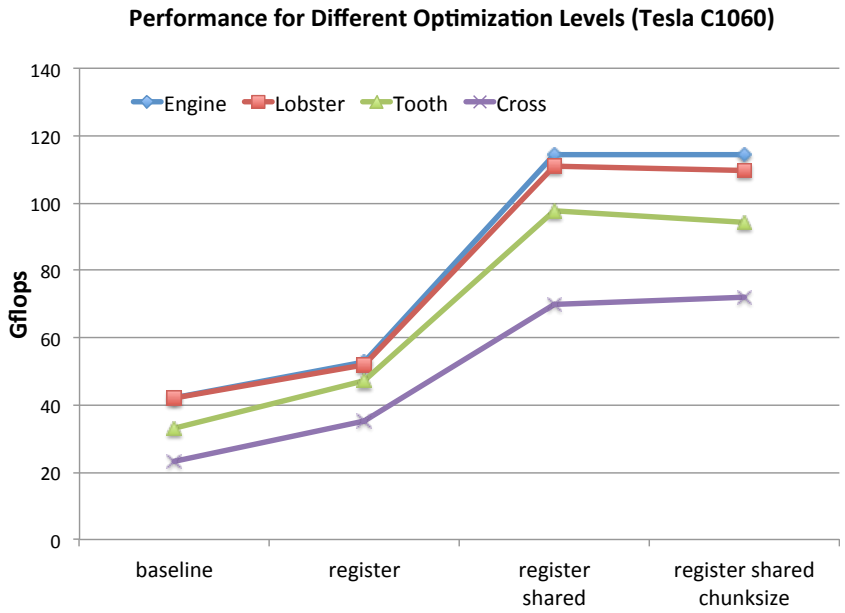


Figure 7.15: Impact of the Mint optimizer on the performance, running on the Tesla C1060. The best performance is achieved when register and shared memory are used.

and shared memory optimizations for four different datasets is 36% and 13% on the C1060 and C2050, respectively. When the register optimizer is combined with the shared memory optimization, shown as register-shared, the optimizer delivers 2.6-3.0 and 1.5-1.75 times the performance of the baseline variants on the C1060 and C2050, respectively.

The right side of Fig. 7.16 shows the results when we enable the *-preferL1* compiler option, which favors larger L1 cache on the C2050. In fact the best performance is achieved when register, shared memory, and large L1 cache are all used together. In the AWP-ODC simulation we showed that favoring L1 cache improved performance but shared memory optimization generally degraded performance on the C2050. However, in the Harris algorithm, there is only one stencil array. As a result, 16KB shared memory is sufficient. Hence, shared memory halves the global memory accesses, which we will discuss shortly. Solely using the *-preferL1* option improves the baseline performance by 23%. When combined with the register optimizer, the cache improves the performance by 55% over the baseline. When we add the shared memory support, there is a 110% performance improvement over the baseline.

The on-chip memory optimizer lets a thread block load a tile of data and ghost cells into shared memory. Each thread is responsible for a single load from global memory into shared

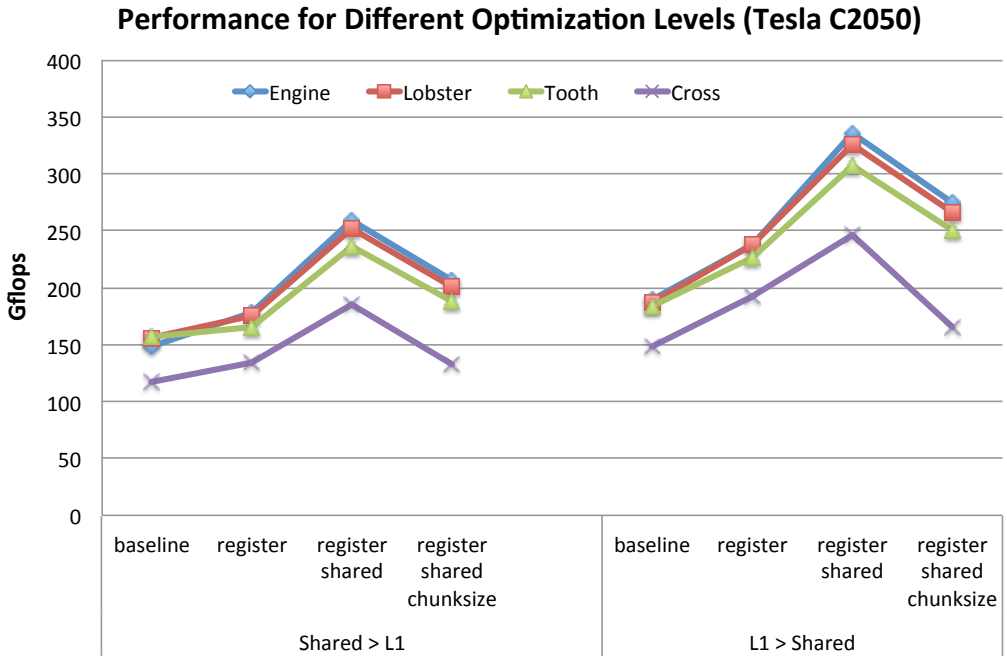


Figure 7.16: Performance impact of the Mint optimizer on the performance on the Tesla C2050. The best performance is achieved when register, shared memory, and large L1 cache are used. Shared > L1 refers to a larger shared memory (48KB) on the C2050 and L1 > Shared refers to a larger L1 cache (48KB).

memory with some threads also loading ghost cells. A thread block reads a tile and neighboring ghost cells on each side into shared memory as illustrated in Fig. 7.17 for 5×5×5 patches. If a thread is assigned to compute the Harris score for the black point, then it needs a 5×5 area covering the convolution window. We would need six more such tiles to cover the ghost cells in the z-dimension because the stencil expands to 3 elements on each side of the z-dimension. The optimizer keeps two tiles for the slowest varying dimension in addition to the center tile in shared memory. The rest of the references go through global memory. With a 5×5×5 convolution window, the optimizer finds 450 shareable references between threads and 135 ghost cells for a 16×16×1 tile. As a result, by using shared memory, the optimizer eliminates 450 references per thread out of 750 memory references. A CUDA thread still performs the remaining 300 references via global memory, where the L1 cache shows its benefit. The reduction in the memory references reflects itself in the results and drastically improves the performance on both devices.

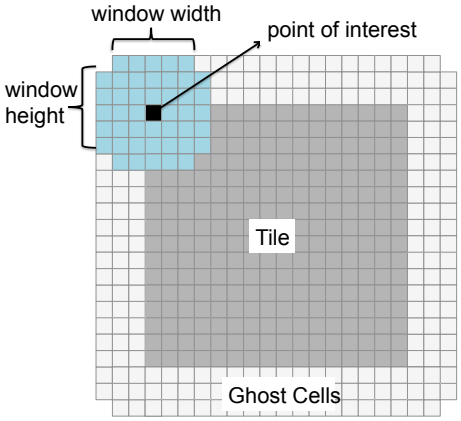


Figure 7.17: A tile and its respective ghost cells in shared memory. The block point is the point of interest. The 5×5 region around the black point shows the coverage of the Gaussian convolution.

The reuse of data in shared memory can be increased further through the `chunksize` clause. The programmer can easily assign more work to a thread by setting a chunking factor. In the example provided in Table 7.6, the chunksize is set to 64 in the outer loop. That is, a thread computes 64 Harris scores in the slowest varying dimension of a 16×16×64 tile. The translator inserts a loop in the generated kernel so that each thread computes more scores. The chunksize optimization is not effective for this algorithm however. The main reason is that assigning more elements to a thread increases the register usage per thread. Since the number of registers is limited on the device, this optimization leads to fewer concurrent thread blocks. On the C1060, performance remains the same when the `chunksize` clause is used. On the C2050, however, performance drops dramatically. Without the chunksize optimization, the number of registers used by a thread on the C2050 is 63, which is the physical device limit. Applying chunksize optimization leads to register spilling onto local memory, resulting in a performance penalty.

7.2.9 Summary

Advanced visualization algorithms are typically computationally intensive and highly data parallel which make them attractive candidates for GPU architectures. Although we studied a single feature detection algorithm, convolution is widely used in computer vision applications. As opposed to the AWP-ODC simulation, we did not implement a hand-optimized version of the algorithm because Mint already enabled us to realize real time performance goals in 3D

images, which previously had been intractable on conventional hardware solutions. We were highly content with the outcome. Our users had no incentive to write the hand-coded version.

Moreover, the GPU acceleration provided a real-time performance without sacrificing accuracy. A large convolution window result in more accurate solution but longer running times. For example, on the 8 Nehalem cores, it takes over 1.76 sec to detect the features for the Engine dataset when the Gaussian convolution uses $5\times5\times5$ patches. To achieve a real-time performance we have to shrink the convolution window to $3\times3\times3$. Since all the Mint-generated kernels run under 1 sec for an $5\times5\times5$ window, Mint achieves real-time feature selection with high accuracy.

Another benefit of using Mint is productivity of the programmer. We obtained real-time performance at a modest cost of inserting 5 lines of Mint pragmas into the original C implementation of the Harris interest point algorithm. Compared to the 389 lines of the original code, this is negligible. Moreover, the programmer can easily change the convolution size in the input program and regenerate the CUDA code with Mint. It would be cumbersome for the programmer to implement the CUDA variants of the algorithm using different convolution sizes because each variant requires a different number of ghost cell loads and sharing between threads. The translator automatically determines the communication between threads with the help of the stencil analyzer in the pre-analysis step and applies optimizations accordingly.

7.3 Aliev-Panfilov Model of Cardiac Excitation

7.3.1 Background

Numerical simulations play a vital role in health sciences and biomedical engineering and can be used in clinical diagnosis and treatment. The Aliev-Panfilov model [AP96] is a simple model for signal propagation in cardiac tissue, and accounts for complex behavior such as how spiral waves break up and form elaborate patterns, as illustrated in Fig.7.18. Spiral waves can lead to life threatening situations such as ventricular fibrillation. Even though the model is simple with only 2 state variables, the simulation is computationally expensive at high resolutions. Thus, there is an incentive to accelerate the simulation in order to improve turnaround time. GPUs have been an effective means of accelerating cardiac simulations [LMB10, LMB09], including the Aliev-Panfilov model. Owing to their small size, GPUs can be integrated in biomedical devices in clinical settings such as hospitals. This potential motivated us to apply Mint to the Aliev-Panfilov model.

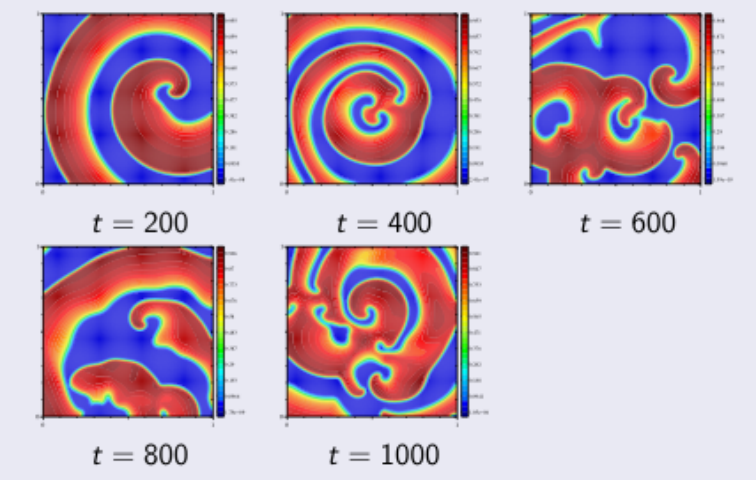


Figure 7.18: Spiral wave formation and breakup over time. Image Courtesy to Xing Cai.

7.3.2 The Aliev-Panfilov Model

The Aliev-Panfilov model is a reaction-diffusion system [Bri86]. The simulation has two state variables. The first corresponds to the transmembrane potential E, while the second represents the recovery of the tissue R. The model reads

$$\frac{\partial E}{\partial t} = \delta \nabla^2 E - kE(E-a)(E-1) - Er \tag{7.17}$$

$$\frac{\partial R}{\partial t} = -[\varepsilon + \frac{\mu_1 R}{\mu_2 + E}] + [E + kE(E-b-1)], \tag{7.18}$$

where the chosen model parameters are $\mu_1 = 0.07$, $\mu_2 = 0.3$, $k = 8$, $\varepsilon = 0.01$, $b = 0.1$ and $\delta = 5x10^{-5}$.

The system can be decomposed into a partial differential equation (PDE) and a system of two ordinary differential equations (ODEs). The ODEs describe the kinetics of reactions occurring at every point in space, and the PDE describes the spatial diffusion of reactants. In the model, the reactions are the cellular exchanges of various ions across the cell membrane during the cellular electrical impulse. Two numerical methods for solving the Aliev-Panfilov model were thoroughly discussed in Hanslien et al. [HAT+11]. In this work, we implemented a finite difference solver for the PDE part and a first-order explicit method for the ODE system.

7.3.3 Stencil Structure and Computational Requirements

The following code fragment shows the inner most loop body for the PDE and ODE solver. The PDE solver uses a finite-difference method. The solver sweeps over a uniformly spaced mesh, updating the voltage E according to weighted contributions from the four nearest neighboring positions in space. The stencil operation is also illustrated in Fig. 7.19.

PDE solver: For each i, j in E:

$$\begin{aligned} E_t(i,j) &= E_{t-1}(i,j) + \alpha * (E_{t-1}(i+1,j) + E_{t-1}(i-1,j) \\ &\quad + E_{t-1}(i,j+1) + E_{t-1}(i,j-1) \\ &\quad - 4 * E_{t-1}(i,j)) \end{aligned}$$

ODE solvers: For each i, j in E and R:

$$\begin{aligned} E_t(i,j) &= E_t t(i,j) - dt * (k * E_t(i,j) * (E_t(i,j) - a) * (E_t(i,j) - 1) \\ &\quad + E_t(i,j) * R_t(i,j)) \\ R_t(i,j) &= R_t(i,j) + dt * (\varepsilon + \mu_1 * R_t(i,j) / (E_t(i,j) + \mu_2)) \\ &\quad * (-R_t(i,j) - k * E_t(i,j) * (E_t(i,j) - b - 1)) \end{aligned}$$

We compute the time integration parameters, α and dt, in the initialization step. Regardless of the precision arithmetic used in the simulation, we always use double precision values to compute these parameters to ensure that there is no loss of precision.

Due to the data dependencies in the PDE solver, the PDE solver requires ghost cell communication with the neighboring threads. The ODE solver is computationally expensive and but is trivially parallizable. It is possible to fuse the PDE and ODEs into one nested-loop. This allows us to reuse the data that is brought into the on-chip memory.

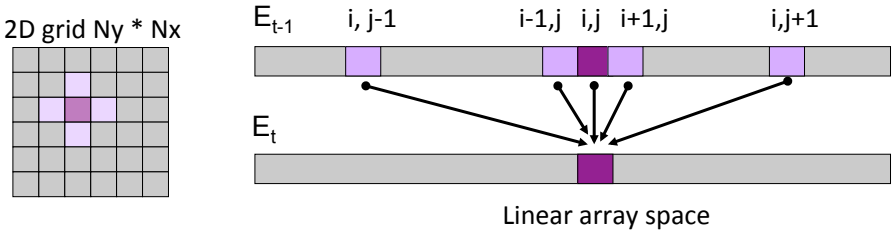


Figure 7.19: The PDE solver updates the voltage E according to weighted contributions from the four nearest neighboring positions in space using 5-pt stencil.

The kernel requires three 2-dimensional grids to store E, R, and E_{prev}, the previous value of E. For a single data point calculation, the PDE solver reads 5 elements from the memory and writes back 1. The ODE solver reads 2 elements and writes back 2 elements. If we assume that once an element read stays in on-chip memory, then the number of distinct loads is 2 for each element in the grid, namely Eprev(i,j) and R(i,j). The number of distinct memory locations written likewise is 2, E(i,j) and R(i,j). As a result, there are ideally 4 memory accesses per grid element in one iteration.

Table 7.9: Instruction mix in the PDE and ODE solvers. Madd: Fused multiply-add. *Madd contains two operations but is executed in a single instruction.

	PDE	ODE 1	ODE 2	Total
Add	3	2	3	8
Multiply	-	3	2	5
Madd*	2	2	3	7
Division	-	-	1	1
Total	7	9	12	28

The Aliev-Panfilov kernel is rich in floating point operations. The kernel has 7 madd (fused multiply-add) instructions, 8 add, 5 multiply and a division operation, totaling 28 flops

Table 7.10: Mint implementation of the Aliev-Panfilov model

```
#pragma mint copy(E, toDevice, (N + 2),(N + 2))
#pragma mint copy(E_prev, toDevice, (N + 2),(N + 2))
#pragma mint copy(R, toDevice, (N + 2),(N + 2))
#pragma mint parallel
{
  while(t < T){
    t = t + dt;
#pragma mint for nest(1) tile(256)
    for(int i = 1; i < (N + 1); i++) {
      E_prev[i][0] = E_prev[i][2];
      E_prev[i][(N + 1)] = E_prev[i][(N - 1)];
      E_prev[0][i] = E_prev[2][i];
      E_prev[(N + 1)][i] = E_prev[(N - 1)][i];
    }
#pragma mint for nest(all) tile(16,16)
    for(int j = 1; j < (N + 1); j++){
      for(int i = 1; i < (N + 1); i++){
        E[j][i]=E_prev[j][i] + alpha * (E_prev[j][i+1]+ E_prev[j][i-1] - 4 * E_prev[j][i] +
                                        E_prev[j + 1][i]+ E_prev[j - 1][i]);
        E[j][i]=E[j][i] - dt*(kk * E[j][i] * (E[j][i]-a) * (E[j][i]-1) + E[j][i] * R[j][i]);

        R[j][i]=R[j][i] + dt * (epsilon + M1 * R[j][i] / (E[j][i] + M2)) *
                              (-R[j][i] - kk * E[j][i] * (E[j][i] - b - 1));
       }
     }
   double** tmp = E; E = E_prev; E_prev = tmp;
  }
} //end of parallel
#pragma mint copy(E_prev, fromDevice, (N + 2),(N + 2))
#pragma mint copy(R, fromDevice, (N + 2),(N + 2))
```

(Table 7.9). Since the kernel makes 4 memory accesses per data point, the flop:word ratio is 7 (28/4). However, the flop:word ratios of the testbed devices are much higher than that of the Aliev-Panfilov (see Fig 6.1 in Chapter 6), and so the kernels are memory bandwidth-limited. On the other hand, we computed the ratios based on the madd throughput. The costly division operation in the kernel lowers the flop:word ratio of the devices. We expect that the double precision variant of the kernel on the Tesla C1060 will be compute-bound since its flop:word ratio for double precision is 8.1.

Table 7.11: Comparing Gflop/s rates of different implementations of the Aliev-Panfilov Model in both single and double precision for a 2D mesh size $4K \times 4K$. Hand-CUDA indicates the performance of a manually implemented and optimized version. Mint indicates the performance of the Mint-generated code when the compiler optimizations are enabled.

Gflops	**Intel Xeon E5530**		**Tesla C1060**		**Tesla C2050**	
$N^2 = 4K \times 4K$	Serial	OpenMP(8)	Mint	Hand-CUDA	Mint	Hand-CUDA
Single Prec.	6.89	23.47	59.51	124.48	140.05	149.48
Double Prec.	3.31	12.68	20.98	25.69	58.62	69.15

7.3.4 Mint Implementation

The Mint program that implements the PDE and ODE solvers of the Aliev-Panfilov method appears in Table 7.10. Lines 1-3 perform data transfers from the host memory to device memory. The loop in Line 9 is a boundary condition loop that updates the boundaries by mirroring the inner region to the boundary region. It is annotated with the Mint `for` directive, which is supplemented with `nest(1)` and `tile(256)`. The Mint translator will generate a CUDA kernel with 1 dimensional thread blocks with size of 256. This program fuses the PDE and ODE solvers into doubly-nested loop in Lines 16-17. The nested loop body will become a CUDA kernel, executed by 2 dimensional thread blocks. Note that we did not annotate the loop with the `chunksize` clause because chunking in 2 dimension is not supported yet. The compiler will notify the programmer if the clause is used. Lastly, Lines 29-30 copy the simulation results back to the host memory.

7.3.5 Performance Results

Table-7.11 compares the Gflop rates for the Aliev-Panfilov model in both single and double precision for a 4K by 4K mesh. The simulation ran for one unit of simulated time, which corresponds to 3544 iterations. The table reports performance for various implementations running on 3 hardware testbeds. The implementations are: hand-coded OpenMP (*OpenMP*), Mint-generated CUDA (*Mint*) and aggressively hand-optimized CUDA (*Hand-CUDA*). The hand-coded variant is the same as that in earlier performance studies of the Aliev-Panfilov method [UCB10].

The multicore implementation running on the Triton cluster does not scale well beyond

4 cores due to bandwidth limitations of this memory intensive kernel. In fact, OpenMP(4) yields nearly the same performance as OpenMP(8). The Mint-generated variants outperform the multicore versions and get close to the performance of hand-optimized CUDA. The Mint code in double precision achieves 85% of the performance of the hand optimized code on both devices. However, in single precision, Mint lags behind the Hand-CUDA version on the C1060. This is because Mint does not yet include the chunksize optimization in 2D kernels, which we have successfully used in 3D stencils. The Hand-CUDA implements this optimization. In [UCB10] we reported that Hand-CUDA achieves 13.3% of the single precision peak performance of the C1060, nearly saturating the off-chip bandwidth. Once the chunksize optimization in 2D has been implemented, we expect performance to increase significantly. Even though we report the results of the hand-coded CUDA for the C2050, we would like to add that the code is not hand-tuned for the Fermi architecture.

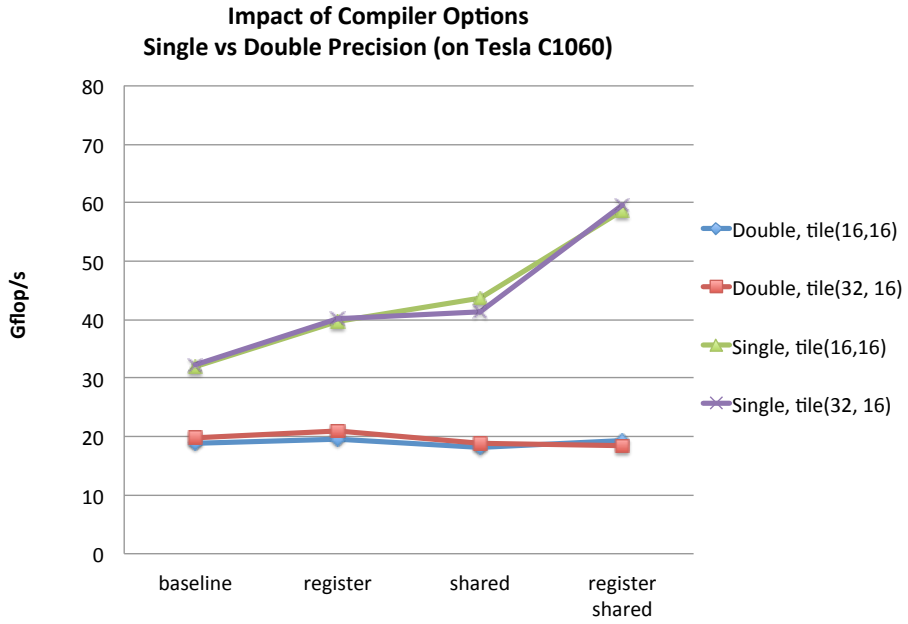


Figure 7.20: Effect of the Mint compiler options on the Aliev-Panfilov method for an input size N=4K. Double indicates double precision. Single indicates single precision.

7.3.6 Performance Tuning with Compiler Options

Fig. 7.20 shows the impact of compiler options on the Tesla C1060 both for single and double precision. We experimented with 2 different tile sizes; (16 $\times$ 16) and (32 $\times$ 16). How-

ever, we did not observe any considerable advantage of choosing one. On the C1060, the Mint optimizer improves baseline performance by a factor of 2 in single precision. Since the optimizer focuses on the memory locality, it cannot help us with the double precision computation on the C1060; the relatively slow double precision arithmetic rate (x8 slower than in single precision), renders this application computation bound. The single precision version, though, benefited from both shared memory and register optimizations because the kernel is memory bandwidth-limited.

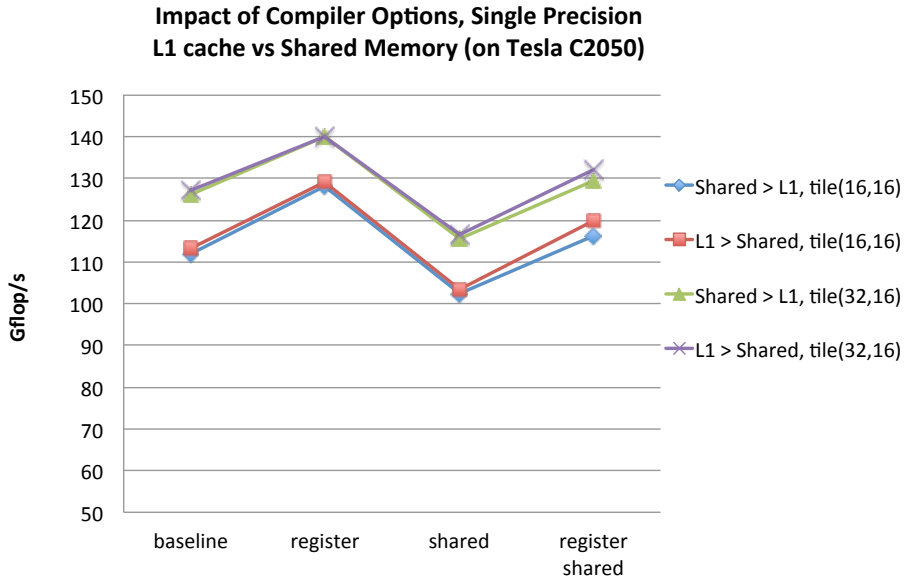


Figure 7.21: Effect of the Mint compiler options on the Aliev-Panfilov method for an input size N=4K. The results are for single precision. *L1* > *Shared* corresponds to favoring a larger cache. *Shared* > *L1* corresponds to favoring a larger shared memory.

Fig. 7.21 shows the single precision performance on the C2050. The shared memory optimization again was not helpful and the best performance is achieved for a tile size of 32 by 16 with the *register* optimizer. The 20 Gflop/s performance difference between the two tiles sizes is consistent for the different compiler flags. Since preferring a larger cache over a larger shared memory did not make any difference in the performance and all the kernels run at 100% occupancy, we believe that the 20 Gflop/s difference is contributed by the cache line size (128 bytes) on the Fermi architecture. Single precision arithmetic enables 32 threads (a warp) to execute concurrently. A smaller tile size than 32 in x-dimension can not fully occupy the load/store units.

Fig 7.22 shows the double precision performance on the C2050. The best performance is attained with a tile size of 16 $\times$ 16 and the *-preferL1* option. For this configuration, using *-register* only and *-register -shared* together give the same performance. The 16 $\times$ 16 thread block size works best for the C2050 because the device can execute up to 16 double precision operations per stream multiprocessor per clock, which is half of the single precision rate. We profiled the register variants for the two different tile sizes. The 32 $\times$ 16 tile size enables 16 active warps in a stream multiprocessor and executes with 33% device occupancy. On the other hand, the 16 $\times$ 16 tile, requiring the same amount of registers per thread as the 32 $\times$ 16, enables 24 active warps with 50% device occupancy. Thus, 16 $\times$ 16 tiles provide more concurrency on the device, leading to higher performance.

For the 16 $\times$ 16 tile size, the *preferL1* variants perform 12% better than when *preferL1* is not used. Using a larger L1 cache improves the hit rate for the kernel even for the variants that use shared memory. The amount of shared memory used by the kernel is small, thus, 16KB is sufficient. The hit rate for the configuration *(16 $\times$ 16 tile, preferL1, register and shared)* is 31%, resulting in 72.6 GB/s memory throughput. The same configuration without *preferL1* gives only 64.5 GB/s throughput with a 15% hit rate.

7.3.7 Summary

We have accelerated the Aliev-Panfilov system using the Mint programming model. On the Tesla C1060, the Mint optimizer improves the baseline performance by a factor of 2 in single precision by utilizing shared memory and registers. We have not yet implemented the chunking optimization for 2D kernels. Once this feature has been implemented, we expect to close the performance gap with the hand-coded version. The double precision code did not enjoy the same performance improvements on the 200-series because the kernel is compute-bound on the Tesla C1060 and our optimizer focuses on reduction in memory traffic. On the C2050, the benefit of the optimizer is modest, permitting up to a 20% improvement.

7.4 Conclusion

In this chapter, we applied the Mint programming model and its compiler to three different applications. The first is a large application modeling ground fault disruption. The second is a computer visualization algorithm, and the last arises in cardiac electrophysiology simulation. In cases where hand-coded implementations are available, we verified that Mint delivered perfor-

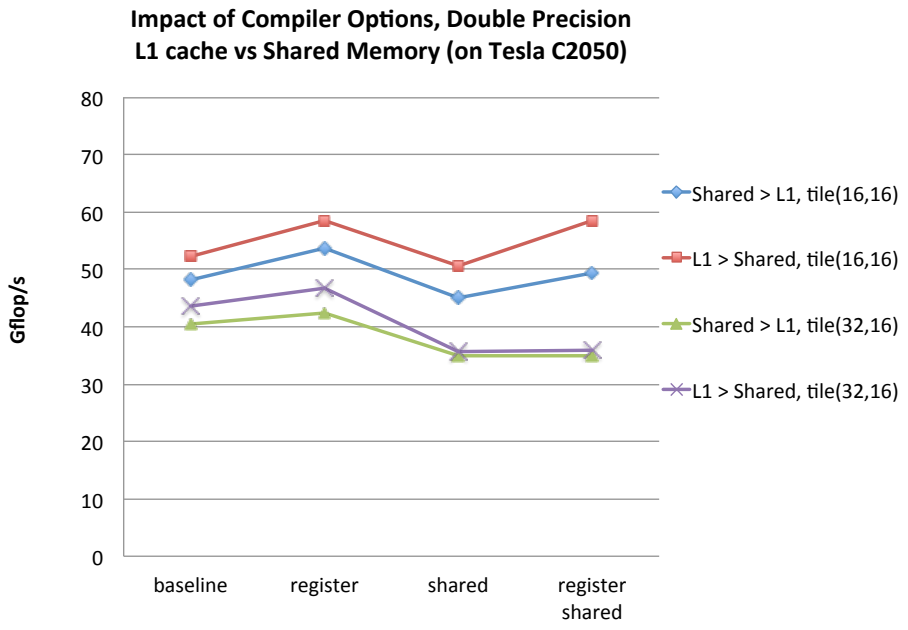


Figure 7.22: Effect of the Mint compiler options on the Aliev-Panfilov method for an input size N=4K. The results are for double precision. *L1 > Shared* favors a larger cache. *Shared > L1* favors a larger shared memory.

mance that was competitive. For the seismic modeling, the Mint generated code realized about 80% of the performance of the hand-coded CUDA. For the cardiac simulation, Mint achieved 70.4% and 83.2% of the performance of the hand-coded CUDA in single and double precision arithmetic, respectively. Mint enabled the visualization algorithm to realize real time performance goals in 3D images, which previously had been intractable on conventional hardware. Therefore, we did not feel the need to implement the hand-coded version.

On the Tesla 1060 device, the Mint compiler significantly improves the performance by utilizing shared memory, register and chunking optimizations. However, the results on the Tesla C2050 suggests the need for further enhancements to the compiler. Generally, the shared memory optimizer did not provide an advantage over the register optimizer and the *preferL1* option. The main contributing factor is that references to shared memory are not cached either in L1 or L2. Thus, the kernels favoring larger shared memory experience more cache misses because L1 is configured to be small. Others reported similar findings [SAT+11] and avoided using shared memory in their applications.

On the C2050, the L1 cache configuration requires experimentation to find the setting

that gives the best performance. Employing a larger cache benefits the baseline variant because some of the global memory references can be cached in L1. If the kernel's shared memory usage is small, or, the kernel is not limited by the shared memory, then a larger L1 should improve performance. In such cases, a larger L1 does not affect the device occupancy but helps the cache hit rate.

Both the shared memory and chunking optimizers increase the number of registers used by a thread, which can be counterproductive. On the C2050, chunking degraded performance in most cases. This is mainly because Fermi limits the number of registers per thread to 63, half that of the 200-series of GPUs. This leaves very little room for the compiler to successfully implement the optimization by using registers. The consequence is typically register spilling to local memory. The C2050 performance results support the argument that further investigation is needed to master the device and findings should be integrated in the translator. In addition, an auto-tuning tool can assist the programmer and the compiler to prune the search space.

Acknowledgements

Section 7.1 in this chapter is based on the material as it partly appears in Computing Science and Engineering Journal 2012 with the title "Accelerating an Earthquake Simulation with a C- to-CUDA Translator" by Jun Zhou, Yifeng Cui, Xing Cai and Scott B. Baden. Section 7.2 in this chapter is based on the material as it partly appears in Proceedings of the 4th Workshop on Emerging Applications and Many-core Architecture 2011, with the title "Auto-optimization of a Feature Selection Algorithm" by Han Suk Kim, Jurgen Schulze and Scott B. Baden. Section 7.3 in this chapter is based on the material as it partly appears in State of the Art in Scientific and Parallel Computing Workshop 2010 with the title "Optimizing the Aliev-Panfilov Model of Cardiac Excitation on Heterogeneous Systems" by Xing Cai and Scott B. Baden. I was the primary investigator and author of these three papers.

Chapter 8

Future Work and Conclusion

8.1 Limitations and Future Work

In this section, we discuss the limitation of the Mint programming model and the Mint compiler. We also describe how we extend both to address current limitations.

8.1.1 Multi-GPU Platforms

The Mint model is currently designed to utilize a single accelerator under the control of a single host thread. Thus, we implemented the model targeting one GPU. Extending Mint to support multiple accelerators (e.g. multiple-GPUs) is important because currently the device memory on the accelerator has only a couple gigabytes of memory, limiting the size of a simulation. In order to support multiple accelerators, the model should allow another level of thread hierarchy, which executes on the host. Each of these host threads will be responsible for controlling one accelerator. Generating code for multiple accelerators will require even a more complex analysis to handle possible data dependencies between computational phases. In particular, the translator will have to include a message passing layer to exchange data between hosts residing on different compute nodes. On the other hand, the translator analyzes ghost cell region for shared memory optimization. This analysis can be extended to include the domain decomposition of the mesh across multiple accelerators and ghost cell exchange during the communication phase.

A GPU can communicate with another GPU through the host processor, resulting in significant cross-GPU latencies. Nvidia GPUDirect 2.0 supports transfers directly between GPUs and NUMA-style direct access to GPU memory from other GPUs. These bring up another in-

teresting problem that motives us to explore the use of non-blocking communication. The Mint translator currently supports only synchronous transfers, although CUDA can perform asynchronous data transfers. This issue should be addressed along with multi-GPU support since non-blocking communication is crucial to hiding the latency.

8.1.2 Targeting Other Platforms

Accelerators can be integrated with the host CPU in the same package, sharing main memory with the host. As shown in Fig. 8.1, such integration removes the PCIe bus between the host processor and accelerator, which is currently a severe bottleneck. We still need the data motion primitives because the memory is partitioned, but the copy runs at the memory speed as opposed to the PCI bus. AMD's Accelerated Processor Unit(APU) is integrated on the chip with the Opteron cores is an example of such design. Any extensions to the Mint programming model have to address the data locality issue between CPU cores and accelerator cores, because each has a dedicated cache or on-chip memory which is not coherent with the other. We are investigating the performance and programming impact of such an architectural design.

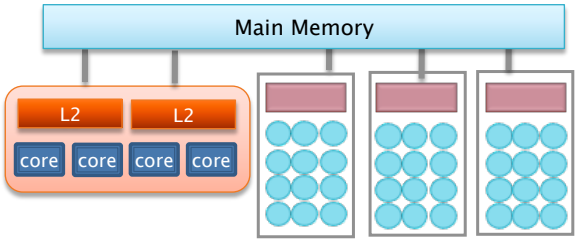


Figure 8.1: Integrated accelerator on the chip with the host cores. All cores share main memory but the memory is partitioned between the host and accelerator.

We have implemented the translator and optimizer for the Nvidia GPUs and generated CUDA code. OpenCL [Opeb] is the open standard for the parallel programming of heterogenous platforms. It has been adopted by Intel, AMD, IBM, Nvidia and ARM. Even though the current performance of OpenCL lags behind CUDA, because of increasing support from vendors, its performance is expected to improve to the point where it will be competitive with CUDA. Intel released the OpenCL implementation for the next generation Ivy Bridge chip architecture and AMD released the OpenCL-driven Accelerated Parallel Processing (APP) Software Development Kit for its ATI graphics cards. The execution and memory model of OpenCL is very similar to CUDA. Thus, the Mint interface does not require changes to target the OpenCL enabled GPUs but the compiler has to be reengineered to generate OpenCL code.

8.1.3 Extending Mint for Intel MIC

The upcoming Intel MIC (Many Integrated Core) architecture, code-named Knights Corner, will be the accelerator in the Stampede Supercomputer [Sup] at the Texas Advanced Computing Center. The MIC will have 50 x86 cores each of which is capable of running four hardware threads and executes 512-bit wide SIMD instructions. The MIC is connected to the host processor with a PCIe card like a GPU. Differently than other GPU accelerators, the MIC cores have a private L1 cache and shared L2 cache that are both coherent with other cores. The MIC accelerator allows many parallel programming models such as OpenMP, MPI, Intel Cilk [BJK+95], thread building blocks [Rei].

To be able generate an optimized MIC code for the stencil methods, we can still employ some of the Mint directives but need to extend the interface, and retarget our compiler. We can easily map the Mint execution model to the MIC execution model. The model will launch asynchronous parallel kernels on the accelerator by the host and offload the computation to the accelerator. Existing data transfer primitives can be used to transfer data between the host and accelerator memory. The Mint compiler for the MIC would take an annotated C source and generate multi-threaded code that either uses thread build blocks or pthreads.

The Mint for-loop clauses and compiler options, however, have to be extended to support the MIC because the memory model on the MIC is different than the one on the current GPUs. The Intel MIC is based on cache hierarchy as opposed to software-managed memory system on the GPUs. Mint's `tile` clause can be used for cache blocking, partitioning the input grid into tiles to improve data locality. Since there are typically more tiles than there are cores, a group of tiles can be assigned to a single core. How this assignment is carried out can be parameterized in the Mint interface, for example, tiles can be assigned to cores in a round-robin fashion. We may not need the `chunksize` clause because the MIC threads are coarser than CUDA threads. On the other hand, we may introduce a clause for register blocking [DMV+08] which subdivides a tile into smaller blocks and unrolls each small block. Lastly, to hide memory latency, the extended Mint interface may include a compiler flag or a for-loop clause to specify software perfecting distance.

Implementing the Mint compiler and optimizer to generate code for the MIC architecture and AMD GPUs is important for wider adoption of the Mint model. The optimization strategies of stencil methods are essentially the same but the implementation of the optimizer exhibits differences based on the target architecture. As we support more platforms, we can introduce a compiler flag, (e.g. `-offload=[mic|apu|cuda]` which sets the target architecture and lets the

compiler optimize for each target.

8.1.4 Performance Modeling and Tuning

Throughout this thesis, we compared the performance of the Mint-generated code with the hand-coded CUDA whenever possible. However, it is unrealistic and impractical to implement every application by hand in order to assess the quality of the compiler generated code. Performance modeling is of great importance to guide the optimizations. Datta et. al [DKW+09] modeled the performance of the stencil methods on cache-based systems. Others have developed a performance modeling tool for GPUs that predicts the application performance based on the floating point intensity, memory bandwidth requirement and thread divergence characteristics of an application [HK09]. Carrington et al. [CTO+11] proposed a performance model based on *idioms* for hardware accelerators. An idiom is a pattern of computation and memory access pattern that may occur within an application. The model predict the speedup of the idioms if those idioms were to be run on accelerators. We can apply these techniques to our performance modeling to predict the performance of the stencil applications.

Based on our experience with the 200 and 400 series of GPUs, we have gained insights into the default values of the tile size and chunksize clauses in addition to the compiler options that yield good performance. However, because of the presence of L1 cache on the Tesla C2050 (based on 400-series of GPUs), the results were less predictable. When a programmer encounters a new device, they have to experiment with different clauses and compiler options to tune the performance. Hence, the best tile size, chunksize and combination of compiler options do not only depend on the device, but the application as well. An auto-tuner can assist the programmer and suggest the combinations of the parameters that yield the best performance. Such tool will explore the optimization space for a particular application on a particular device and give feedback to the programmer. Williams explored the auto-tuning techniques in his dissertation [Wil08] on multicore processors. The OpenMPC [LE10] compiler comes with an optimization space pruner to assist the programmer. The same principles can be applied to develop a tool on top of our translator. This addition to the translator will be worth considering given the continuing evolution of GPU architectures.

8.1.5 Domain-Specific Translators

An obvious limitation of our translator is that it is domain-specific. Mint targets stencil computations and our optimizations are specific to this problem domain. The benefit of this ap-

Table 8.1: Performance of non-stencil Kernels. MatMat: Matrix-Matrix Multiplication.

	MatMat	Transpose	Reduction
Mint	2.5 Gflop/s	5.6 GB/s	13.5 GB/s
Nvidia SDK	357 Gflop/s	66.3 GB/s	83.8 GB/s

proach outweighs the disadvantages of the limitations. We can incorporate application semantics into our compiler, resulting in improved performance. Thus, even for the non-stencil applications, we believe that Mint provides two benefits: first, it can be used in instructional settings to teach students the concept of accelerators and familiarize them with CUDA programming. Second, an expert can modify the Mint-generated code to obtain high performance for non-stencil kernels. Mint relieves the programmer from handling tedious tasks such as data transfer and thread management.

Table 8.1 shows the results for a couple non-stencil kernels: a matrix-matrix multiplication, matrix transpose and reduction on a vector. The CUDA variants are highly optimized implementations that come with the Nvidia Software Development Kit (SDK). Compared to the SDK variants, the performance of Mint is poor since Mint does not perform any domain-specific optimizations on the these kernels. On the other hand, it successfully generates the CUDA code.

By following the Mint example we can build similar domain-specific optimizers to target other domains. Since we want to avoid the increase in optimization space, we are concerned with not losing the semantic knowledge in the application, thus we can get hints from the programmer. For example, a programmer can identify the motifs in the program and annotate code sections as `#stencil` or `#sparse` to indicate that the stencil optimizer or sparse linear algebra optimizer should be used to transform that code section. This piece of information passed to the compiler would greatly simplify code analysis and improve the quality of the generated code. The current interface of Mint is sufficient for stencil methods but in order to support multiple domains we would need to add new directives to the interface.

8.1.6 Compiler Limitations

Though we have implemented it for standard C the Mint interface is language neutral. The same interface can be adopted to different front-end languages such as Fortran or Python. Both Fortran and Python are easier to handle than C because the C pointers complicate analysis due to aliasing. We enforced some restrictions on C to simplify our analysis such as how arrays

are referenced and allocated in Mint. Our translator can perform subscript analysis on multi-dimensional array references only (e.g., A[i][j]). It cannot analyze “flattened" array references. For example, when determining stencil structures, Mint may incorrectly disqualify a computation as not expressing a stencil pattern, thus Mint will not optimize it. In addition, the `copy` directive cannot determine the shapes of dynamically allocated arrays. We require that the programmer contiguously allocate the memory for an array so that the copy from the source array to the destination requires only single transfer. If memory is not contiguous, the copy results in an undefined behavior, possibly in a segmentation fault. Future work including runtime support and dynamic analysis will lift these restrictions.

Another limitation of the compiler concerns the scope of a `parallel` region. The `for`-loop directive should be enclosed statically within a `parallel` region. In other words, we cannot branch out from a parallel region into a routine containing a for-loop directive. Mint requires the programmer to inline all the nested-loops that need to be accelerated into a parallel region. In future work, we would like to support *orphaned* loops. Orphaning allows for-loops to appear outside the lexical extent of a parallel region and is particularly useful for parallelizing bulky loops.

8.2 Conclusion

We proposed the Mint programming model to address the programming issue of a system, comprising a traditional multicore processor (host) and a massively parallel multicore (accelerator). With the non-expert programmers in mind, we based our model on compiler directives. Directives do not require substantial recoding and can be ignored by a standard compiler. Thus, it is possible to maintain a single code base for both host and accelerator variants that contain different kinds of annotations.

To annotate programs, Mint employs five directives that are sufficient to accelerate diverse stencil-based applications. The five directives are: 1) `parallel` to indicate the start of an accelerated region, 2) `for` to mark the succeeding nested loop for work-sharing, 3) `barrier`, 4) `single` to synchronize, and handle serial sections within a parallel region. 5) `copy` to handle data transfers between host and accelerator. While the last pragma does not describe how data motion will be carried out, it does expose the separation of host and device address spaces.

We developed a fully automated translation and optimization system that implements the Mint model for GPU-based accelerators. The Mint compiler parallelizes loop-nests, performs data locality optimizations and relives the programmer of a variety of tedious tasks. Mint’s

optimizer is not general-purpose. It targets stencil-based codes that appear in many scientific applications that employ a finite difference method. By restricting the application domain, we are able to achieve a high performance on massively parallel multicore processors.

The Mint compiler has two stages: The first stage is the *Baseline Translator* which transforms C source code with Mint annotations to unoptimized CUDA, generating both device code and host code. The second stage of the Mint compiler is the *optimizer* which performs architecture- and domain-specific optimizations. The output of the translator can be subsequently compiled by `nvcc`, the CUDA C compiler.

The Mint programmer can optimize code for acceleration incrementally with modest programming effort. The code optimization is accomplished by supplementing a Mint `for` pragma with clauses and by specifying various compiler options. Both clauses and compiler options hide significant performance programming, enabling non-experts to tune the code without entailing disruptive reprogramming. The `nest(#)` clause specifies the depth of the data parallel `for`-loop nest. There is a similar mechanism in OpenMP that allows parallelization of perfectly nested loops without specifying nested parallel regions, called `collapse(#)`[1]. An OpenMP compiler collapses loops to form a single loop and then splits the iteration space among threads. However, Mint performs multi-dimensional partitioning of the iteration space which is crucial in finite difference stencils to occupying the device effectively. The remaining two clauses govern workload decomposition across threads and hence cores. The `tile` clause specifies how the iteration space of a loop nest is to be subdivided into *tiles*. A tile corresponds to the number of data points computed by a thread block. The `chunksize` clause allows the programmer to manage the granularity of the mapping of workload to threads. This clause is particularly helpful when combined with on-chip memory optimizations because it enables data re-use.

Mint provides a few compiler options to manage data locality for stencil methods, often in conjunction with the `for`-loop clauses. The optimizations utilize on-chip memory resources (register, shared memory and cache) to improve memory bandwidth. The *register* option enables the Mint register optimizer, which takes advantage of the large register file residing on the device. The optimizer places frequently accessed array references into registers. Since the content of a register is visible to one thread only, this option improves the accesses to the central point of a stencil, but not the neighboring points that are shared by other threads. For that purpose, Mint provides the shared memory optimizer triggered by the *shared* flag. The shared memory optimizer detects the sharable references among threads and chooses the most frequently accessed

[1]This feature is added to OpenMP 3.0 to target multicore architectures.

array(s) to place in shared memory. This optimization is particularly beneficial for the stencil computation because of the high degree of sharing between threads. The final option concerns cache. Certain accelerators (such as Fermi-based GPUs) come with a configurable on-chip memory as software- or hardware-managed. Mint allows the programmer to favor hardware-managed memory over software-managed memory with a compiler option, namely *preferL1*.

Both the register and shared memory optimizations rely on the stencil analyzer, an indispensable part of the Mint optimizer. This component of the optimizer analyzes the stencil pattern and ghost cell structure appearing in the code to select variables for shared memory and registers. We implemented an algorithm to rank arrays referenced in a kernel based on their potential reduction in global memory accesses when placed in on-chip memory. The ranking algorithm also takes into account the amount of shared memory needed by an array. The algorithm picks the arrays that maximize the total reduction in global memory references and minimize the shared memory usage.

The combination of compiler options and loop clauses yielding best performance differs from device to device as well as from application to application. Both register and shared memory optimizations potentially increase the register usage and lower the device occupancy. In some cases, applying these optimizations may be counterproductive because the performance gain resulting from a reduction in global memory references might not compensate the performance loss resulting from lower device occupancy. A programmer has to implement many variants of the same program manually to find the best configuration, which is cumbersome. Shared memory optimization is particularly challenging due to the management of ghost cells. In this context, a source-to-source translator becomes handy. Mint gives the programmer the flexibility to explore different variants of the same program with reasonable programming effort. The programmer experiments different optimization flags and `for` loop clauses and incrementally tunes the performance.

We showed the effectiveness of Mint on commonly used stencil kernels. Mint achieves about 80% of the performance obtained by aggressively hand-optimized code on the Nvidia C1060 and C2050. Mint is not only useful in the laboratory, but also in enabling acceleration of whole applications such as the earthquake modeling, AWP-ODC. We inserted only 6 Mint for-directives to annotate the most time consuming loops in the simulation. Compared to the two hundred lines of the original computation code. The results are encouraging – Mint-generated code achieved up to 83% of the performance of hand optimized CUDA code. Mint delivers high performance on a variety of other applications including a computer vision algorithm, the

3D Harris interest point detection algorithm. Mint enabled the user to realize interactive interest point detection in volume datasets without having to learn CUDA. The last application we studied was the Aliev-Panfilov system which models signal propagation in the heart. Mint achieved 70% and 83% of the performance of the hand-coded CUDA in single and double precision arithmetic, respectively. In addition to these three applications we ourselves studied, some researchers from Simula Research Laboratory have adopted Mint and successfully ported 3D Parallel Marching Method into CUDA by using Mint. A quote from their paper [GSC12]:

"Using the automated C-to-CUDA code translator Mint, we obtained a CUDA implementation without manual GPU programming. The GPU implementation runs approximately 2.4-7 times faster than the fastest multi-threaded version on 24 CPU cores[2]*, giving hope to compute large 3D grids interactively in the future.*

Lastly, we would like Mint to be useful to other researchers and have made the compiler available to the public. The Mint compiler is built on top of the ROSE compiler framework [QMPS02] from Lawrence Livermore National Laboratory. As of November 2011, Mint is integrated into the ROSE compiler and distributed along with ROSE. Alternatively, the Mint source code is available online for download for those who already have ROSE installed in their system. We also provide a web portal where users can upload their input file for online translation so that they do not need to install the software. Appendix A describes these three sources in more detail.

[2] 24 AMD magny cores on the Hopper supercomputer.

Appendix A

Mint Source Distribution

The Mint compiler is comprised entirely of open source code and distributed with the BSD license. We made the Mint source available to public in three different ways:

1. Mint is available for download from code.google. We recommend this distribution for experienced users since it is the most up-to-date version of the compiler but the users have to manage the ROSE integration by themselves.

   `http://code.google.com/p/mint-translator/`

2. Mint is distributed along with the ROSE compiler framework. We recommend this distribution for first time users since the Mint source is already merged into ROSE.

   `http://www.rosecompiler.org/`

3. We also provide a web portal where users can upload their input file for online translation so that they do not need to install the software. However, the online translator is not fully featured, only uses the baseline translation without any optimizations.

   `http://ege.ucsd.edu/translate.html`

We encourage users to report any problems, and to make suggestions for enhancements, through our Bugzilla-based bug tracking system: `http://ege.ucsd.edu/bugzilla3/`

Table A.1 shows the Mint source code directory. The *midend* directory includes the baseline translator (7832 lines) and the *optimizer* directory includes the optimizer (4754 lines). The entire source is 12781 lines of C++ code.

Please check the following Mint project website for any updates.

`http://sites.google.com/site/mintmodel/`

Table A.1: Mint source code directory structure. The lines of codes is indicated in parenthesis.

license.txt README rose.mk		
src (12781)	Makefile main.C (80)	
	midend (7832)	
		MintCudaMidend.C (926) LoweringToCuda.C (1582) CudaOutliner.C (950) Reduction.C (460) VariableHandling.C (148) arrayProcessing MintArrayInterface.C (986) memoryManagement CudaMemoryManagement.C (1345) mintPragmas MintPragmas.C (957) mintTools MintOptions.C (195) MintTools.C (283)
	optimizer (4754)	
		CudaOptimizer.C (820) programAnalysis StencilAnalysis.C (733) MintConstantFolding.C (245) OptimizerInterface CudaOptimizerInterface.C (523) OnChipMemoryOptimizer OnChipMemoryOpt.C (1125) GhostCellHandler.C (903) LoopUnroll LoopUnrollOptimizer.C (405)
	types (115)	
		MintTypes.h (115)

Appendix B

Cheat Sheet for Mint Programmers

The Mint compiler can be called with the "mintTranslator" command followed by the input filename. The inputfile must have an .c extension and the path to the inputfile can be absolute or relative path.

```
$ ./mintTranslator [options] input_file.c
```

Possible options are:

`-o | output filename` : specifies output file name.

If not set, Mint will use default filename (mint_inputfile.cu).

`-opt:shared=[# of slots]` : turns on shared memory optimization.

If # of slots is not specified, Mint will use 8 slots.

`-opt:register` : turns on register optimization.

`-opt:preferL1` : favors a larger L1 cache.

`-opt:useSameIndex` : arrays use common indices to reduce the register usage.

Arrays must have the same dimension and size.

`-opt:unroll` : Unrolls short loops and performs constant folding optimization.

We enforce some restrictions on C to simplify our stencil analysis such as how arrays are *allocated* and *referenced* in Mint. Mint can perform subscript analysis on multidimensional array references only (e.g., A[i][j]). It cannot analyze “flattened" array references (e.g, A[i*N+j]). For example, when determining stencil structures, Mint may incorrectly disqualify a computation as not expressing a stencil pattern, thus Mint will not optimize it. Thus, we recommend programmers to use subscript notion on arrays. In addition, the `copy` directive cannot determine

Table B.1: Contiguous memory allocation for a 3D array

```
1  float ***alloc3D(int N, int M, int K)
2  {
3    float ***buffer=NULL;
4    buffer = (float***)malloc(sizeof(float**)* K);
5    assert(buffer);
6
7    float** tempzy = (float**)malloc(sizeof(float*)* K * M);
8    float *tempzyx = (float*)malloc(sizeof(float)* N * M * K );
9
10   for ( int z = 0 ; z < K ; z++, tempzy += M ) {
11     buffer[z] = tempzy;
12     for (int y = 0 ; y < M ; y++, tempzyx += N ) {
13       buffer[z][y] = tempzyx;
14     }
15   }
16   return buffer;
17 }
```

the shapes of dynamically allocated arrays that will be used on the accelerator. Mint requires that the programmer contiguously allocate the memory for an array so that the copy from the source array to the destination array requires only single transfer. If memory is not contiguous, the transfer results in an undefined behavior, possibly in a segmentation fault. Therefore, we recommend that Mint programmers use the subroutine provided in Table B.1 which ensures contiguous memory allocation for 3D arrays, and at the same time allows subscripted array references.

B.1 Mint Interface

A Mint program is a legal C program, annotated with Mint directives. The Mint interface provides 5 compiler directives (pragmas), summarized in Table B.2. For more details, please refer to Chapter 3.

A Mint program contains one or more designated accelerated regions. Each of these regions contains code sections that will execute on the accelerator under the control of the host. Although, all code in the region is not able to run on the accelerator, rather, only *kernels* can. A kernel is typically an annotated work-sharing nested-loop, or infrequently annotated serial region. All other code that is not in the accelerated region runs on the host. There is an implicit synchronization point at the end of each kernel unless the programmer specified otherwise.

In the Mint model, data movement is managed by the compiler with the assistance of

Table B.2: Mint Directives and Supportive Clauses

Directives and Optional Clauses	Description
`#pragma mint parallel` `{ }`	indicates the scope of an accelerated region
`#pragma mint for \`	marks data parallel for-loops
`nest(# | all)`	depth of loop parallelism
`tile(`t_x`, `t_y`, `t_z`)`	partitioning of iteration space
`chunksize(`c_x`, `c_y`, `c_z`)`	workload of a thread in the tile
`reduction(operator:var_name)`	reduction operation on the specified variable.
`nowait`	enables host to resume execution
`#pragma mint copy(src | dst,` `toDevice | fromDevice,` N_x`, `N_y`, `N_z`, ... )`	transfers data between host and accelerator
`#pragma mint barrier`	synchronizes host and accelerator
`#pragma mint single` `{ }`	serial execution on the accelerator

the programmer annotations. The programmer must be aware of the separate address spaces and ensure that data is transferred to the device memory at the entry of a parallel region and transferred back if needed to the host at the exit of a parallel region. Mint takes care of storage allocation and deallocation of the variables on the device. At the exit of a parallel region the allocated memory for the device variables will be freed and the content will be lost. As a result, threads created in different parallel regions cannot communicate through device memory.

Inside a parallel region, if a programmer needs to access device variables on the host, let's say for IO, then she needs to explicitly insert copy primitives to the program. In such cases, Mint conserves the allocated space for variables and their content on the device memory.

On-chip memory on the device is managed by the compiler with the compiler options provided by the programmer. The programmer does not need to know how the on-chip memory works but should be aware of their trade-off. More on-chip memory usage means fewer concurrent threads running on the device. Excessive usage can be counterproductive and diminish performance. Moreover, the limited on-chip memory size may lead to compile or runtime errors.

Appendix C

Mint Tuning Guide for Nvidia GPUs

C.1 Tuning with Clauses

nest: Parallelizing all levels of a loop nest improves inter-thread locality and also leads to vastly improved occupancy. Since Mint currently parallelizes up to triple nested loops, the programmer should use nest(all) with caution.

tile: The best choice of the tile size depends on the application and device. The programmer has to experiment with different configurations. Our recommendation is to choose a tile size multiple of 16 in the x-dimension to ensure aligned memory accesses. However, on the Fermi devices, L1 cache line size is 128 bytes. For single precision arithmetic, multiple of 32 in the x-dimension should be used so that each thread reads 4 bytes of data from a cache line. For double precision arithmetic, multiple of 16 is sufficient because the device can execute up 16 double precision operations per vector core per clock cycle, half of the single precision rate. The tile size in the y-dimension can be chosen smaller than 16 but should be multiple of 2.

If there is no performance difference between two tile size configurations, smaller value should be chosen because a smaller tile size can compensate for the register requirements when the compiler options are used. Fewer threads mean more on-chip memory resources per thread.

chunksize: The value of chunksize is less sensitive to the on-chip resources but the clause itself increases the register usage regardless of its value. We recommend a value multiple of 32 because a small value won't amortize the overhead introduced by the loop. On the 200-series on chip memory and chunking optimizations are crucial to delivering high performance. On the Fermi architecture, performance gain is less predictable because of the presence of cache and requires more experimentation.

C.2 Tuning with Compiler Options

-opt:register: The register optimization generally improves performance on both 200-series and Fermi devices. If the usage of the register option degrades performance, it might be due to the register spilling. Fermi limits the number of registers per thread to 63, which is 127 on the 200-series. The nvcc compile spills registers to local memory when this limit is exceeded. The latency to local memory is as high as global memory. Favoring a larger L1 may cache spilled registers, however, this may lead to contention with other data in the cache.

-opt:shared: A kernel is more amenable to the shared memory optimization, if the stencil-like global memory references are concentrated on few arrays and each with high access frequencies because shared memory saves a high number of references to global memory. On the 200-series of GPUs, the shared memory optimization for stencil methods generally improves performance because there is no cache. On the Fermi devices, shared memory may or may not improve performance depending on the reduction in global memory references. The usage of shared memory increases the demand for registers which can be counterproductive. This is an artifact of the nvcc compiler. Excessive register and shared memory pressure can lower device occupancy to a level where there are not sufficient threads to hide memory latency.

-opt:preferL1: Although we provide some insight into when preferring a larger L1 is more advantageous, we highly suggest experimentation to determine the best cache configuration for a given kernel. A larger L1 cache can be advantageous because it does not affect the device occupancy but improves the cache hit rate. Hence, cache does not increase register pressure in the same way as shared memory does. Employing a larger cache benefits the baseline variant because L1 can cache the global memory accesses.

For a kernel using shared memory, if the 16KB shared memory is sufficient to buffer the stencil arrays, a large L1 cache should be used so that non-stencil arrays, for example coefficient arrays, can be cached. What is sufficient is highly correlated to the device occupancy. If the kernel is not limited by shared memory, but limited by the number of registers or warp size, favoring L1 will improve the performance because the available shared memory suffices.

Bibliography

[ABB+99] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999. 17

[ABC+06] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006. 11

[ADD+09] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180, 2009. 18

[AF05] R. J. Atkin and N. Fox. *An Introduction to the Theory of Elasticity*. Dover Publications, 2005. 98

[AMD11] AMD. *AMD Accelerated Parallel Processing OpenCL Programming Guide*. 2011. 10, 11, 21

[AP96] R. Aliev and A. V. Panfilov. A Simple two-variable model of cardiac excitation. *Chaos, Solions & Fractals*, 7(3):293-301, 1996. 129

[BB09] Francois Bodin and Stephane Bihan. Heterogeneous multicore parallel programming for graphics processing units. *Scientific Programming - Software Development for Multi-core Computing Systems*, 17:325–336, December 2009. 18, 22

[BBC+08] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, Katherine Yelick, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Stephen Keckler, Dean Klein, Peter Kogge, R. Stanley Williams, and Katherine Yelick. Exascale computing study: Technology challenges in achieving exascale systems, 2008. 1, 7

[BGMS97] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997. 17

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM. 141

[Bri86] Nicholas F. Britton. *Reaction-diffusion Equations and Their Application to Biology*. Academic Press, 1986. 129

[BS97] I. N. Bronshtein and K. A. Semendyayev. *Handbook of Mathematics*. Springer-Verlag, New York, NY, USA, 3rd edition, 1997. 16

[CA07] Stephen Craven and Peter Athanas. Examining the viability of FPGA supercomputing. *EURASIP Journal on Embedded Systems*, 2007:13–13, January 2007. 8

[CJvdP07] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007. 18, 27, 29

[Cle] http://www.clearspeed.com/. 8

[CMHM10] Eric S. Chung, Peter A. Milder, James C. Hoe, and Ken Mai. Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs? In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 225–236, Washington, DC, USA, 2010. IEEE Computer Society. 8

[COJ+10] Yifeng Cui, Kim B. Olsen, Thomas H. Jordan, Kwangyoon Lee, Jun Zhou, Patrick Small, Daniel Roten, Geoffrey Ely, Dhabaleswar K. Panda, Amit Chourasia, John Levesque, Steven M. Day, and Philip Maechling. Scalable earthquake simulation on petascale supercomputers. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–20, Washington, DC, USA, 2010. IEEE Computer Society. x, 96, 97, 104

[Col04] Phil Colella. Defining Software Requirements for Scientific Computing, 2004. 11

[COS11] M. Christen and H. Burkhart O. Schenk. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Interational Conference on Parallel and Distributed Computing Systems (IPDPS)*, 2011. 23

[CSB+11] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM Symposium on Principles and Practice*

of Parallel Programming, PPoPP '11, pages 35–46, New York, NY, USA, 2011. ACM. 23

[CTO+11] Laura Carrington, Mustafa M. Tikir, Catherine Olschanowsky, Michael Laurenzano, Joshua Peraza, Allan Snavely, and Stephen Poole. An idiom-finding tool for increasing productivity of accelerators. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 202–212, New York, NY, USA, 2011. ACM. 142

[Cyb] Cybershake. http://epicenter.usc.edu/cmeportal/cybershake.html. 97

[DD07] L.A. Dalguer and S. M. Day. Staggered-grid split-node method for spontaneous rupture simulation. In *Journal of Geophysical Research*, Vol. 112, 2007. 96, 97

[DGM+10] Ron O. Dror, J. P. Grossman, Kenneth M. Mackenzie, Brian Towles, Edmond Chow, John K. Salmon, Cliff Young, Joseph A. Bank, Brannon Batson, Martin M. Deneroff, Jeffrey S. Kuskin, Richard H. Larson, Mark A. Moraes, and David E. Shaw. Exploiting 162-nanosecond end-to-end communication latency on "Anton". In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–12, Washington, DC, USA, 2010. IEEE Computer Society. 8

[DKW+09] Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Review*, 51(1):129, 2009. 142

[DLD+03] William J. Dally, Francois Labonte, Abhishek Das, Patrick Hanrahan, Jung-Ho Ahn, Jayanth Gummaraju, Mattan Erez, Nuwan Jayasena, Ian Buck, Timothy J. Knight, and Ujval J. Kapasi. Merrimac: Supercomputing with streams. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 35–, New York, NY, USA, 2003. ACM. 9

[DMM+10] Anthony Danalis, Gabriel Marin, Collin Mccurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, and Jeffrey S. Vetter. The scalable heterogeneous computing "(SHOC)" benchmark suite. In *in Proc. 3-rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU3*, pages 63–74, 2010. 81

[DMV+08] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 4:1–4:12. IEEE Press, 2008. 12, 35, 141

[DWV+09] K. Datta, S. Williams, V. Volkov, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Auto-tuning the 27-point stencil for multicore. In *iWAPT, 4th International Workshop on Automatic Performance Tuning*, 2009. 61

[FQKYS04] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. GPU cluster for High Performance Computing. In *SC '04: Proceedings of the 2004 ACM/IEEE*

conference on Supercomputing, page 47, Washington, DC, USA, 2004. IEEE Computer Society. 9

[FVS11] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A comprehensive performance comparison of CUDA and OpenCL. In *The 40-th International Conference on Parallel Processing (ICPP'11), Taipei, Taiwan*, September 2011. 21

[GSC12] Tor Gillberg, Mohammed Sourouri, and Xing Cai. A new parallel 3D front propagation algorithm for fast simulation of geological folds. In *Proceedings of the International Conference on Computational Science, ICCS 2012*, Procedia Computer Science. Elsevier, 2012. 147

[HA09] Tianyi David Han and Tarek S. Abdelrahman. hiCUDA: a high-level directive-based language for GPU programming. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 52–61, New York, NY, USA, 2009. ACM. 22

[HAT+11] Monica Hanslien, Robert Artebrant, Aslak Tveito, Glenn Lines, and Xing Cai. Stability of two time-integrators for the Aliev-Panfilov system. *International Journal of Numerical Analysis and Modeling*, Vol 8, No. 3:427–442, 2011. 130

[HK09] Sunpyo Hong and Hyesoon Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 152–163, New York, NY, USA, 2009. ACM. 142

[HS88] C. Harris and M. Stephens. A combined corner and edge detector. In *Proceedings of the 4th Alvey Vision Conference*, pages 147–151, 1988. 117, 119

[iLJE03] Sang ik Lee, Troy A. Johnson, and Rudolf Eigenmann. Cetus - an extensible compiler infrastructure for source-to-source transformation. In *Languages and Compilers for Parallel Computing, 16th Intl. Workshop, College Station, TX, USA, Revised Papers, volume 2958 of LNCS*, pages 539–553, 2003. 40

[Int11] The OpenACC Application Program Interface. *OpenACC 1.0 Specification, http://www.openacc-standard.org/*. 2011. 23

[Jag] Jaguar. Compute cluster. http://www.cray.com/Products/XT/ORNLJaguar.aspx. 97

[JJ88] H. Jeffreys and B. S. Jeffreys. *Methods of Mathematical Physics*. Cambridge University Press, Cambridge, England, 3rd edition, 1988. 16

[KAB+03] Nam Sung Kim, Todd Austin, David Blaauw, Trevor Mudge, Krisztián Flautner, Jie S. Hu, Mary Jane Irwin, Mahmut Kandemir, and Vijaykrishnan Narayanan. Leakage current: Moore's law meets static power. *Computer*, 36:68–75, December 2003. 5

[KDH+05a] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, 2005. 12, 21

[KDH+05b] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM J. Res. Dev.*, 49:589–604, July 2005. 58

[KDK+01] Brucek Khailany, William J. Dally, Ujval J. Kapasi, Peter Mattson, Jinyung Namkoong, John D. Owens, Brian Towles, Andrew Chang, and Scott Rixner. Imagine: Media processing with streams. *IEEE Micro*, 21:35–46, March 2001. 8

[KDS+11] Jens Krueger, David Donofrio, John Shalf, Marghoob Mohiyuddin, Samuel Williams, Leonid Oliker, and Franz-Josef Pfreund. Hardware/software co-design for energy-efficient seismic modeling. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 73:1–73:12, New York, NY, USA, 2011. ACM. 8

[KKH02] Joe Kniss, Gordon Kindlmann, and Charles Hansen. Multidimensional transfer functions for interactive volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):270–285, 2002. 117

[KSC+10] Han Suk Kim, Jürgen P. Schulze, Angela C. Cone, Gina E. Sosinsky, and Maryann E. Martone. Dimensionality reduction on multi-dimensional transfer functions for multi-channel volume data sets. *Information Visualization*, 9(3):167–180, 2010. 117

[KUSB12] Han Suk Kim, Didem Unat, Jürgen P. Schulze, and Scott B. Baden. Interactive data-centric viewpoint selection. *Proceedings SPIE 8294, 829405*, 2012. 117

[LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. 40

[LBS+01] Bill Lorensen, Chandrajit Bajaj, Lisa Sobierajski, Machiraju, and Jinho Lee. Visualization viewpoints: The transfer function bake-off. *IEEE Computer Graphics and Applications*, 21, 2001. 117

[LE10] Seyong Lee and Rudolf Eigenmann. OpenMPC: extended OpenMP programming and tuning for GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11. IEEE Computer Society, 2010. 21, 142

[LKC+10] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *SIGARCH Comput. Archit. News*, 38:451–460, June 2010. ix, 12

[LMB09] F. Lionetti, A McCulloch, and S. B. Baden. GPU accelerated solvers for ODEs describing cardiac membrane equations. In *NVidia GPU Technology Conference*, October 2009. 129

[LMB10] Fred V. Lionetti, Andrew D. McCulloch, and Scott B. Baden. Source-to-source optimization of CUDA C for GPU accelerated cardiac cell modeling. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing*, EuroPar'10, pages 38–49, Berlin, Heidelberg, 2010. Springer-Verlag. 23, 129

[LME09] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 101–110, New York, NY, USA, 2009. ACM. 2, 18, 21, 40, 93

[LQVP09] Chunhua Liao, Daniel J. Quinlan, Richard W. Vuduc, and Thomas Panas. Effective source-to-source outlining to support whole program empirical optimization. In *Languages and Compilers for Parallel Computing, 22nd International Workshop, LCPC 2009*, volume 5898 of *Lecture Notes in Computer Science*, pages 308–322. Springer, 2009. 39

[Mat] http://www.mathworks.com/products/matlab/. 17

[McC95] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995. 80

[Mic09] Paulius Micikevicius. 3D finite difference computation on GPUs using CUDA. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84. ACM, 2009. 35, 58

[MK10] David Michéa and Dimitri Komatitsch. Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards. *Geophysical Journal International*, 182(1):389–402, 2010. 58

[MO03] Carey Marcinkovich and Kim Olsen. On the implementation of perfectly matched layers in a three-dimensional fourth-order velocity-stress finite difference scheme. In *Journal of Geophysical Research*, Vol. 108, B5, 2276, 2003. 98

[Moo00] Gordon E. Moore. Readings in computer architecture. cramming more components onto integrated circuits. pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000. 5

[Mud01] Trevor Mudge. Power: A first-class architectural design constraint. *Computer*, 34(4):52–58, April 2001. 5

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with "CUDA". *Queue*, 6:40–53, March 2008. 10, 18, 19

[Nvi] Nvidia. Nvidia visual profiler. http://developer.nvidia.com/nvidia-visual-profiler. 90

[Nvi07] Nvidia. CUDA CUFFT Library, 2007. 18

[Nvi10a] Nvidia. *CUDA programming guide 3.2*. 2010. 48

[Nvi10b] Nvidia. *Tuning CUDA Applications for Fermi v1.3*. 2010. 89, 112

[ODM+06] K.B. Olsen, S. M. Day, J. B. Minster, Y. Cui, A. Chourasia, M. Faerman, R. Moore, P. Maechling, and Jordan T. H. Strong shaking in Los Angeles expected from southern San Andreas earthquake. In *Geophysical Research Letters*, Vol. 33, 2006. 96

[OHL07] Kunle Olukotun, Lance Hammond, and James Laudon. *iChip Multiprocessor Architecture: Techniques to Improve Throughput and Latency*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2007. 5

[Ols94] K. B. Olsen. Simulation of three-dimensional wave propagation in the Salt Lake basin. Ph.D Thesis, 1994. 95

[opea] http://code.google.com/p/opencurrent/. 23

[Opeb] http://www.khronos.org/opencl/. 21, 140

[PLS+00] Hanspeter Pfister, Bill Lorensen, Will Schroeder, Chandrajit Bajaj, and Gordon Kindlmann. The transfer function bake-off. In *VIS '00: Proceedings of the conference on Visualization '00*, pages 523–526, Los Alamitos, CA, USA, 2000. IEEE Computer Society Press. 117

[Pop] http://www.psc.edu/machines/sgi/altix/pople.php. 19

[QMPS02] Daniel J. Quinlan, Brian Miller, Bobby Philip, and Markus Schordan. Treating a user-defined parallel library as a domain-specific language. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, pages 324–. IEEE Computer Society, 2002. 38, 147

[Rei] James Reinders. Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. page 334. O'Reilly Media. 141

[ros] Rose. http://www.rosecompiler.org. 38

[RT00] Gabriel Rivera and Chau-Wen Tseng. Tiling optimizations for 3D scientific computations. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, SC '00. IEEE Computer Society, 2000. 58

[SAT+11] Takashi Shimokawabe, Takayuki Aoki, Tomohiro Takaki, Toshio Endo, Akinori Yamanaka, Naoya Maruyama, Akira Nukada, and Satoshi Matsuoka. Peta-scale phase-field simulation for dendritic solidification on the TSUBAME 2.0 supercomputer. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 3:1–3:11, New York, NY, USA, 2011. ACM. 137

[SDM11] 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. 1

[Str04] John C. Strikwerda. *Finite Difference Schemes and Partial Differential Equations, 2nd Edition*. SIAM, 2004. 15

[Sup] Stampede Supercomputer. http://www.tacc.utexas.edu/news/press-releases/2011/stampede. 141

[TEE+97] Pl Oi Ts, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17:12–19, 1997. 8

[The11] The Khronos OpenCL Working Group. OpenCL - The open standard for parallel programming for heterogeneous systems. http://www.khronos.org/opencl, February 2011. 21

[Til] http://www.tilera.com/. 8

[TKM+02] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The "Raw" microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002. 8

[Top] http://www.top500.org/. 6, 8

[Tri] Triton. Triton compute cluster. http://tritonresource.sdsc.edu/cluster.php. 80

[UCB10] Didem Unat, Xing Cai, and Scott Baden. Optimizing the Aliev-Panfilov model of cardiac excitation on heterogeneous systems. *Para 2010: State of the Art in Scientific and Parallel Computing*, June 6-9 2010. 133, 134

[UCB11] Didem Unat, Xing Cai, and Scott B. Baden. Mint: realizing CUDA performance in 3D stencil methods with annotated C. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 214–224, New York, NY, USA, 2011. ACM. 2

[VD08] 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. IEEE Press, 2008. 12, 14, 59, 89, 108

[Wil08] Samuel Webb Williams. *Auto-tuning Performance on Multicore Computers*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008. 142

[WKKR99] Christian Weiß, Wolfgang Karl, Markus Kowarschik, and Ulrich Rüde. Memory characteristics of iterative methods. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, Supercomputing '99, New York, NY, USA, 1999. ACM. 16

[WM95] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Computer Architecture News*, 23(1):20–24, 1995. 8

[WOCS11] Samuel Williams, Leonid Oliker, Jonathan Carter, and John Shalf. Extracting ultra-scale "Lattice Boltzmann" performance via hierarchical and distributed autotuning. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 55:1–55:12, New York, NY, USA, 2011. ACM. 93

[Wol10a] Michael Wolfe. Implementing the PGI Accelerator model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 43–50. ACM, 2010. 2

[Wol10b] Michael Wolfe. Implementing the PGI Accelerator model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 43–50, 2010. 18, 22

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

[WSO+06] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd Conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM. 12

[WSO+07] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. Scientific computing kernels on the Cell processor. *Int. J. Parallel Program.*, 35:263–298, June 2007. 58

[Yel08] Katherine Yelick. Programming model challenges for managing massive concurrency. In *SC'08 Workshop on Power Efficiency and the Path to Exascale Computing*, 2008. 19