UiO : Department of Informatics
University of Oslo

LLVM supported source-to-source translation

Translation from annotated C/C++ to CUDA C/C++

Niklas Jacobsen

Master's Thesis Autumn 2016

LLVM supported source-to-source translation

Niklas Jacobsen

1st November 2016

Abstract

The need for computing power is constantly increasing and this has popularized the utilization of specialized computing units, such as a Graphics Processing Unit (GPU), for general-purpose computing. General-purpose GPUs provide tremendous computing power with relatively low power-consumption. The addition of a GPU to a system makes the computing architecture heterogeneous and more complex. In order to fully take advantage of the computing power of both the Central Processing Unit (CPU) and the GPU, applications must be specifically programmed for such a system composition. Programming heterogeneous systems is complex, time-consuming and often requires detailed knowledge of the underlying hardware architecture.

In an effort to minimize the effort needed to utilize the GPU, Didem Unat created the Mint programming model[61, chapter 3] and the Mint source-to-source translator[61, chapter 4]. Even though the Mint translator is very good, we recognize potential for improvements.

In this thesis we present Spearmint, our source-to-source translator that accepts Mint-annotated C and C++ code and generates CUDA C/C++ code that can be run on a CPU-GPU system. The source-to-source translator presented in this thesis is based on the LLVM compiler infrastructure and supports all common operating systems.

We implemented an optimization that utilizes Nvidia's Kepler architecture's read-only data cache. Our comparisons show that Spearmint delivers comparable or higher performance than the Mint translator for three commonly used stencil computations on one of Nvidia's Kepler GPUs. We also found that the Spearmint-generated CUDA code is shorter, less involved and resembles the input code more than the Mint-generated code.

Acknowledgement

First and foremost, I would like to thank my supervisor Xing Cai for his patience, guidance and invaluable feedback throughout the entire process. I would also like to thank my supervisor Mohammed Sourouri for his help with testing and valuable input on the technical aspects of my thesis.

My thanks also go to my fellow students on the 9th floor. Thank you for all the coffee breaks, entertaining card games, long lunches and interesting conversations.

Finally, I would like to thank my family and friends for their encouragement, support and understanding throughout my studies.

Contents

1 Introduction **1**
1.1 Thesis Goals 2
1.2 Structure of Thesis 3

2 Motivation and Background **5**
2.1 Trends in Computing Architecture 5
2.1.1 General-Purpose Multicore Processors 7
2.1.2 Massively Parallel Accelerators 8
2.1.3 Programming Graphics Processing Unit 9
2.1.4 Many Integrated Core Architecture 13
2.2 Application Characteristics 14
2.3 Parallel Programming Models 16
2.3.1 Nvidia's CUDA 18
2.3.2 OpenMP 20
2.3.3 OpenACC 21
2.3.4 OpenCL 21
2.4 Summary 22

3 The Mint Programming Model **25**
3.1 The Model 25
3.1.1 Mint's Execution Model 27
3.1.2 Mint's Memory Model 27
3.2 The Mint Interface 29
3.2.1 The Parallel Directive 29
3.2.2 The For Directive 30
3.2.3 The Copy Directive 32
3.2.4 The Barrier Directive 33
3.2.5 The Single Directive 33

3.3 The Mint Translator 34
3.4 Summary 36

4 **The LLVM Compiler Infrastructure** **39**
4.1 LLVM 39
4.2 Clang 42
4.2.1 LibTooling 44
4.2.2 LibRewrite 45
4.3 Summary 46

5 **Spearmint Source-to-Source Translator** **47**
5.1 Adding Mint Directives to Clang 48
5.1.1 Modifying Clang's Parser 49
5.1.2 Modifying Clang's Semantic Library 49
5.1.3 Modifying Clang's AST Library 50
5.2 Creating Spearmint with LibTooling 51
5.3 Memory Management 53
5.3.1 Error Handling 56
5.4 Kernel Management 57
5.4.1 General Analysis 57
5.4.2 Analysing the memory accesses 59
5.4.3 Replacing Externally Declared Arrays 61
5.4.4 Inserting Forward Declarations of Kernels 61
5.4.5 Inserting Kernel Configuration Code 62
5.4.6 Inserting Kernel Execution Code 63
5.4.7 Kernel Creation 64
5.5 Read-Only Cache Optimization 67
5.6 Spearmint's Compiler Options 69
5.7 Installing Spearmint 70
5.8 Summary 72

6 **Evaluating the Implementation** **75**
6.1 The Testing Environment 75
6.2 The Test Codes 76
6.2.1 Tile and Chunksize Configurations 78
6.3 Baseline Performance 79
6.3.1 Baseline performance of the 3D 7-pt Heat Stencil code 80

6.3.2 Baseline Performance of the 3D 7-pt Heat Stencil with Variable Coefficients 81
6.3.3 Baseline Performance of the 3D 19-pt Poisson Code . 83
6.4 Prefer L1 Cache Configuration 84
6.5 Comparing Spearmint's Read-only Cache Optimization with Mint's Optimizations 85
6.5.1 Optimizing the 7-pt Heat Laplace Operator 86
6.5.2 Optimizing the 3D 7-pt Heat with Variable Coefficients 88
6.5.3 Optimizing the 3D 19-pt Poisson Code 90
6.6 Code Quality and Usability 92
6.7 Summary 94

7 Related Work 97

8 Future Work and Conclusion 101
8.1 Future work 101
8.1.1 Further Optimizations 101
8.1.2 General Improvements 102
8.2 Conclusion 103

List of Figures

2.1 Abstract view of a GPU connected to a traditional multicore CPU system. 10
2.2 Two example stencils. a) shows a 7-point 3D stencil and b) shows a 19-point 3D stencil. 15
2.3 Thread and memory hierarchy in CUDA. 19

3.1 The Mint translator's reuse of data loaded into shared memory while chunking in the z-dimension. The values in a z-plane is reused for each iteration. 35

4.1 An example of a traditional three-phase design. 40
4.2 The modularity of the three-phase design due to LLVM's IR. 42

5.1 Overview of Spearmint's design and its control-flow. 48

6.1 Comparison of the baseline performance of the 3D Heat 7-pt Laplace operator between Mint and Spearmint 81
6.2 Comparison of the baseline performance of the 3D Heat 7-pt variable coefficient between Mint and Spearmint. 82
6.3 Comparison of the baseline performance of the 3D Poisson 19-pt code between Mint and Spearmint. 84
6.4 Comparison of the performance of the 3D Heat 7-pt Laplace operator between optimized Mint and Spearmint. 86
6.5 Comparison of the performance of the 3D Heat 7-pt variable coefficient between optimized Mint and Spearmint. 89
6.6 Comparison of the performance of the 3D Poisson 19-pt code between optimized Mint and Spearmint. 91

Listings

4.1 A simple example of LLVM's IR 41
4.2 The corresponding code from listing 4.1 in C. 41
5.1 Excerpt from a Mint annotated 7-pt 3D stencil code implemented in C++. 51
5.2 Recommended memory allocation code in C++ for code that is to be translated by Spearmint. 54
5.3 Recommended memory deallocation code in C++ for memory allocated with the code provided in listing 5.2. . . . 55
5.4 Example memory-copy code generated by Spearmint. 56
5.5 Example output from the analysis phase. 59
5.6 An example replacement of an externally declared array. The code shown is a result of translating line 17 in listing 5.1. 61
5.7 An example of a forward-declaration of a kernel generated by Spearmint. 62
5.8 An example of kernel configuration code generated by Spearmint for the 7-pt 3D stencil code in listing 5.1 62
5.9 An example kernel execution call generated by Spearmint for the 7-pt 3D stencil code in listing 5.1 64
5.10 A CUDA kernel generated by Spearmint for the 7-pt 3D stencil code in listing 5.1. 66
5.11 The resulting computation for line 23 in the 7-pt 3D stencil code in listing 5.10 with read-only cache optimization. 68

List of Tables

3.1 Summary of the Mint directives. 37

4.1 Clang's libraries and tools. 43

5.1 Spearmint's translation options. 70

6.1 A summary of the stencil codes used for testing. 77

Acronyms

API Application Programming Interface

AST Abstract Syntax Tree

CFG Context-free Grammar

CPU Central Processing Unit

ECC Error-Correcting Code

FLOPS Floating Point Operations per Second

FPGA Field-Programmable Gate array

IR Internal Representation

MIC Many Integrated Core

MPI Message Passing Interface

OpenACC Open Accelerators

OpenCL Open Computing Language

OpenMP Open Multi Processing

GPGPU General-Purpose computing on Graphics Processing Unit

GPU Graphics Processing Unit

RAM Random-Access Memory

SIMD Single Instruction Multiple Data

SIMT Single Instruction Multiple Threads

SM Streaming Multiprocessor

Chapter 1

Introduction

There is an ever increasing demand for faster computation. Performance improvements allow for more complex computations and operations applied to larger datasets than ever before. Fields such as machine learning, computer vision, physics simulations, etc. allow scientists to develop novel applications that solve hard problems. Examples of such applications are earthquake simulations[16, 63], image recognition[9], biological simulations[6]. These kinds of computation often require large amounts of computing power and this need for high performance drives the development of computing architectures.

Traditionally, supercomputers consisting of hundreds or thousands of nodes, each with their own CPU was used for such computational tasks. Supercomputers are expensive to acquire, maintain and consume vast amounts of electricity. However, new developments in computing architectures offer higher performance and at the same time lower power consumption. This is achieved by adding specialized computing units with a high degree of parallelism, such as a GPU, to a supercomputer's nodes.

Adding specialized computing units or *accelerators* to the supercomputers makes the node's architecture heterogeneous and this has implications for programming. Programs that utilize the full computing capability of such heterogeneous architectures must handle intra-node memory transfers, communication between the host-processor and the accelerator as well as contain accelerator-specific programming.

The added complexity that comes with the newer computing architectures creates a demand for programming models and tools that simplify

the job of the programmer, in order to efficiently develop applications for such heterogeneous architectures.

In this thesis we present one such tool; a source-to-source translator that utilizes the Mint programming model[61, chapter 3]. The Mint model was developed by Didem Unat and she also developed a Mint-conforming source-to-source translator, which is also named Mint[61, chapter 4]. The Mint translator is based on the Rose compiler architecture[51] and users have reported difficulties installing it as a result. For this and other reasons we chose to base our translator on the LLVM compiler infrastructure[23], LLVM, and why we chose it, will be described in chapter 4. The Mint translator accepts Mint-annotated C code and generates optimized CUDA C code for Nvidia's Fermi architecture. Nvidia has since the Fermi architecture developed the Kepler architecture and we want our tool to specifically target it. Additionally, there is a wish to support Mint-annotated C++ code, as well as Mint-annotated C code. The source-to-source translator presented in this thesis accepts Mint-annotated C and C++ code and generates optimized CUDA C/C++ code for Nvidia's Kepler architecture. In order to limit our efforts, our translator only generate CUDA code that utilize a single GPU on the system it is executed on.

1.1 Thesis Goals

The major goal for this thesis is to create a source-to-source translator based on the LLVM compiler infrastructure. The translator must be able to translate sequential Mint annotated C and C++ code to CUDA C/C++ that can be executed on Nvidia's CUDA GPUs. The translator should be able to:

- Generate CUDA code that delivers high performance on Nvidia's CUDA GPUs based on Nvidia's Kepler architecture.
- Be easy to install and use by non-expert programmers.
- Be used as a learning tool where the user learns by examining the generated code with respect to the input code.

- Accept a combination of Mint annotations and OpenMP annotations in the input code, allowing easily achieved parallelism on both the CPU and the GPU.

1.2 Structure of Thesis

Chapter 2: *Motivation and Background*

This chapter provides the motivation and background for this thesis. The chapter describes the trends in computing architecture and processor technologies and the implications these trends have for application development. The chapter also describes the application characteristics of the applications we focus on in this thesis and some of the more popular development tools, programming models and languages used in current systems.

Chapter 3: *The Mint Programming Model*

Describes the Mint programming model. Mint is a high-level programming model that allows the programmer to guide a Mint-conforming compiler/translator in its process of generating parallelized code. The source-to-source translator presented in this thesis implements the Mint programming model.

Chapter 4: *The LLVM Compiler Infrastructure*

This chapter presents the LLVM compiler infrastructure. The LLVM compiler infrastructure provides several libraries and tools that perform compilation-related tasks. The source-to-source translator presented in this thesis is based on the LLVM compiler infrastructure.

Chapter 5: *Spearmint Source-to-Source Translator*

Describes the implementation of our source-to-source translator, Spearmint. The chapter illustrates the implementation of the translation process by showing the translation of a 7-pt three-dimensional stencil code, step-by-step.

Chapter 6: *Evaluating the Implementation*

This chapter evaluates the performance of Spearmint, our source-to-source translator. We evaluate the achieved results when translating

three commonly used stencil codes. We compare the results achieved by Spearmint with code generated by the Mint translator. We also provide a brief discussion of the two translators' usability and the quality of the generated code.

Chapter 7: *Related Work*
In this chapter we present a brief overview of related work.

Chapter 8: *Future Work and Conclusion*
Chapter 8 concludes this thesis, describes our contributions and discusses whether we have achieved the goals for this thesis. The chapter also presents our thoughts on future work.

Chapter 2

Motivation and Background

This chapter provides background information about the trends in computing architecture and processor technologies, and discusses why massive parallelism is increasing. We provide a discussion of why the trend of massive parallelism creates the need for new software tools as a result of these trends. In section 2.2 we describe the characteristics of the applications that we focus on in this thesis. We also describe some of the more popular parallel programming models, development tools and languages used in current systems.

2.1 Trends in Computing Architecture

Moore's law states that there will be a doubling of the number of transistors on a densely integrated circuit approximately every two years[39]. This prediction has held true for several decades. However, fulfilling this rate of improvement is no longer feasible due to energy consumption and physical restrictions. The semiconductor industry therefore turned to other means of improving the performance of the CPU. The solution entailed grouping several compute cores on the same processor die, thus starting the era of multicore processors. 80% of all supercomputers on the Top500 supercomputer list[59] were using multicore processors in 2012[61, p. 6]. Comparatively, all of the supercomputers on the Top500 supercomputer list use multicore processors today[59]. Multicore CPUs have the ability to perform several tasks in parallel. In fact, in order to fully utilize them, performing tasks in parallel are a requisite.

The parallel trend in supercomputers has persisted, and in more recent times the trend has expanded to building supercomputers with *accelerators* or *co-processors*. An accelerator is typically a single-chip massively parallel processor consisting of hundreds or thousands of computing cores. Such accelerators provide Tera Floating Point Operations per Second (TFLOPS)[1] performance and higher energy efficiency compared to the multicore CPU. A GPU is a typical example of one such accelerator. Nvidia's Tesla K20 GPU has a peak theoretical single precision floating point performance of 3.52 TFLOPS and a peak theoretical double precision floating point performance of 1.17 TFLOPS, which dramatically outperforms modern multicore CPUs[57]. On the top500 list from June 2016, 18.8% of all supercomputers utilize an accelerator/co-processor, and this trend seems to be continuing[59].

Modern heterogeneous supercomputers usually consist of a high number of nodes. Each node is typically a standalone machine containing its own multicore processor, Random-Access Memory (RAM), hard drives, I/O peripherals and in recent times, one or more accelerators. These nodes are connected to each other through an interconnect that facilitates inter-node communication. The interconnect technology used varies but some of the more prevalent ones are Infiniband and Ethernet[59]. The architecture of the typical supercomputer adds another layer of parallelism. On the topmost level the application divides its workload among the nodes. On the node level each node divides its assigned work among the processor cores and the co-processor, if any. On the lowest level each core performs its work iteratively or parallel through the use of Single Instruction Multiple Data (SIMD) instructions. The division of labour adds complexity for the programmers caused by inter-node communication, inter-core communication and communication between the host-processor and the accelerator. There are several research initiatives working to simplify the programming of heterogeneous supercomputers that take the different kinds of communication required into account while at the same time delivering high performance[37, 52, 53]. However, in this thesis we restrict our field of research to a single host-system with a single accelerator.

In order to efficiently program applications that run on this newer system-composition, good tools and programming models are imperative.

[1] Tera floating-point operations per second or 10^{12} FLOPS.

This thesis aims to contribute in the development of such development-tools. Further in this section we will give a brief introduction to general-purpose multicore processors and massively parallel accelerators with a focus on GPUs, which are increasingly being used in the field of high-performance computing.

2.1.1 General-Purpose Multicore Processors

The newer multicore processors have the same capabilities as the earlier single-core processors. Multicore processors support interrupts, instruction level parallelism, branching, arithmetic instructions and context switches which are needed in a modern operating system. The main difference is that the multicore processors contain several cores on the same die as the earlier single-core processors. Multicore processors offer both task-parallelism where the cores are executing different tasks in parallel and data-parallelism where each core is performing the same task on different data.

Modern multicore processors often have support for SIMD. SIMD is a type of data-parallelism, where the same instruction is applied to several data-points in parallel. When implementing data-parallelism, the programmer maps the data onto SIMD registers in order to use the SIMD instructions. There exist compiler optimizations that automatically utilize SIMD instructions where applicable, but the compiler is not always capable of generating machine code that produces the most optimal code. Hence there is a need to manually implement code that utilizes SIMD instructions.

In order to utilize the full performance of a multicore processor, the workload should be divided onto each of the computing cores to ensure that all cores are performing work. Failing to properly balance the workload will result in some cores being idle/under-utilized while some cores has too much work leading to sub-optimal performance. There is an upper theoretical limit of how much an application may be sped up by performing tasks in parallel. This theoretical upper-limit is defined by Amdahl's law or Amdahl's argument[50].

Not all applications display a high-degree of parallelism, and some codes require time-consuming redesign in order to expose as much

parallelism as possible[19]. Redesigning for parallelism is often a complex and error prone task. The programmer needs to ensure that the program is thread-safe and does not allow for synchronization errors, dead-locks, race conditions, and other problems that may arise during parallel execution.

As described above there are several challenges relating to programming multicore processors. There exist tools that simplify parallel programming on multicore processors, the most prominent one is Open Multi Processing (OpenMP) and we will discuss OpenMP in subsection 2.3.2.

2.1.2 Massively Parallel Accelerators

Massively parallel accelerators are often placed on its own chip connected to the host system via a memory-bus that handles communication between the host-system and the accelerator. The hardware architecture is dramatically different from the traditional CPU. The accelerators usually contain several hundreds or thousands of specialized compute cores. These cores are far simpler than a core on a multicore processor, and can only perform a subset of the instructions. As a result of the high-level of parallelism these accelerators display, the working frequency of the accelerator is lower, compared to the multicore processor, and as an effect, the accelerator consumes less energy.

The co-processors usually have their own separate memory space on-chip, separated from the rest of the system they are connected to. The separate memory space has impact on how applications that utilize them handle memory. In order for the accelerator to work on some data, device-memory must first be allocated. Then, the data must be copied over the memory-bus from the host-system, to the device-memory allocated on the accelerator. The memory movement described is usually explicitly programmed by the application developer.

The specialized nature of the accelerator usually means that it must be controlled by the host-system. The host-system controls the accelerator by initiating different actions. Initiating a *kernel*, e.g. a sub-routine that performs some computation on the device is an example of one such action[44, p. 7][14, section 2.4]. Other actions the host-system are responsible to initiate are memory transfers and synchronization calls.

As mentioned in section 2.1, a GPU is a typical example of an

accelerator, and as this thesis specifically targets GPU-programming, we will go detailed into its architecture in the next section. We will follow that with a brief discussion of Intel's new Many Integrated Core (MIC) architecture, as that seems to be the GPUs fiercest contender in the context of supercomputers and co-processors.

2.1.3 Programming Graphics Processing Unit

When the GPU was in its infancy, it was designed with the intention of only handling graphics-related computation. For some time it was used for only that purpose, until smart programmers realized that the speed of the GPU could be applied to other problems as well. These programmers reformulated their graphics-unrelated computations as if they were a graphics computation in order to run their computations on the GPU. In doing so they often gained significant performance increases. However, this kind of programming were time-consuming, difficult and required extensive knowledge of the particular GPU the application would run on. When realizing the GPU's potential in solving graphics-unrelated problems the General-Purpose computing on Graphics Processing Unit (GPGPU) programming concept was born. The GPU manufacturers started facilitating GPGPU computations on their hardware and GPGPU programming has become significantly easier since then. However, GPGPU programming is still complex, time-consuming and requires that the programmer has detailed knowledge of the GPU hardware, often on a per-card basis in order to write the most optimized code.

Graphics processing units exist in mainly two forms, either as an integrated part of the host-system, or as a separate card connected to the host system via a memory bus. The latter configuration is more common in cases where high-performance is sought after, as the stand-alone GPUs are generally faster than the embedded ones.

Computations on the GPU are run as *kernels* in a Single Instruction Multiple Threads (SIMT) fashion and a single thread/compute core may be used for computing a single point in the problem space. Applying GPUs on parallel applications results in high parallelism, high arithmetic throughput and high performance.

GPUs are not general-purpose in the same way general-purpose CPUs

are, and are dependent of a host-CPU that acts as a controller, runs the operating system, etc. The programmer handles memory transfers, kernel execution, synchronization calls etc. and these operations are initiated by the host-CPU. The programmer also have to implement the kernels that are run on the GPU.

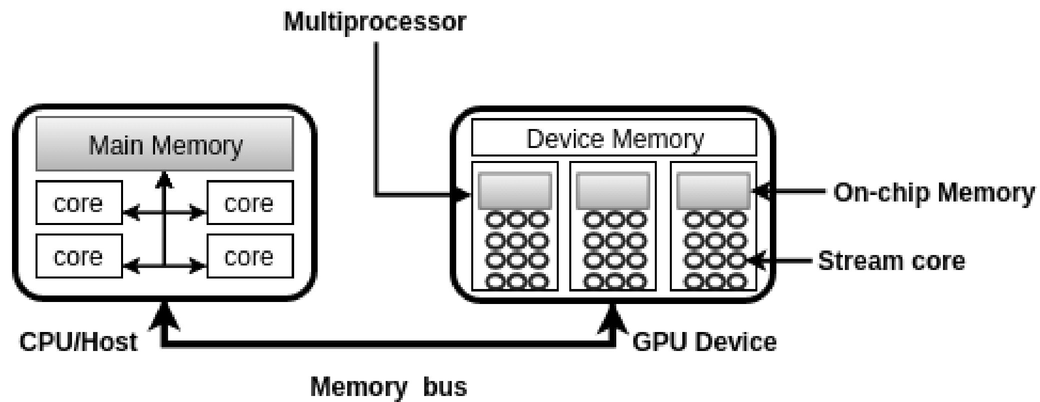

Figure 2.1: Abstract view of a GPU connected to a traditional multicore CPU system.

An abstract view of the GPU hardware can be seen in figure 2.1. Note that the terminology used for the different units on the GPU further on in this text is not vendor specific. The computational units on a GPU are hierarchically divided. The top-most units are the *multiprocessors*[2]. Each multiprocessor has control over hundreds of *stream cores*. Each stream core has its own arithmetic unit and performs the computation assigned to a thread. A stream core/thread is the smallest computational unit and represents the most fine-grained granularity in a GPU application. Several stream cores are grouped together and forms a thread group[3]. A thread group represents the most coarse-grained granularity and computes a part of the entire problem size. Each thread group is dynamically assigned to a multiprocessor by the hardware. One multiprocessor can execute several thread groups simultaneously. The multiprocessor is responsible for the execution of the stream cores in the thread group assigned to it. A thread group is further divided into sub-groups[4], which is the most basic unit of scheduling on the hardware. Each stream core/thread in the sub-group

[2]Nvidia's CUDA refer to a multiprocessor as a Streaming Multiprocessor (SM)
[3]A thread group is referred to as a thread block in Nvidia's CUDA[65].
[4]Referred to as a *warp* in Nvidia hardware or as a *wavefront* in AMD hardware.

performs the same instruction scheduled by the sub-group scheduler. This mode of computations is what is called SIMT.

The size of a sub-group, the number of stream cores per multiprocessor and the number of multiprocessors per GPU vary from manufacturer to manufacturer and is hardware dependent. These metrics often also vary on a card-to-card basis from each manufacturer. The number of threads in a thread group is usually software-configured by the running application through execution-configuration functionality.

The size of the thread group impacts the degree of device *occupancy*. Too small thread groups (smaller than the size of a sub-group) lead to few concurrent threads running in parallel due to under-utilization of the sub-group schedulers, which are a limited resource on each multiprocessor. This is because a sub-group is the smallest unit that may be scheduled and the size of a sub-group is static. This means that a thread group with a size of one thread will be executed as a single sub-group where 31 of the sub-group's threads will be idle, given a sub-group size of 32. Selecting a thread-group size that is a multiple of the sub-group size is therefore preferable.

Too big thread groups that use large amounts of on-chip memory may also decrease occupancy because there are not enough resources to allow execution of several thread groups on a multiprocessor concurrently. Experimenting with different sizes often yield performance benefits. This type of experimentations can be quite time-consuming and incur a big development-cost. In addition, the most optimal configuration varies from GPU to GPU and applications often need to be redesigned and re-optimized in order to achieve maximum performance when switching out the GPU.

The GPU's memory hierarchy is also quite different from the memory hierarchy on a typical CPU. As seen in figure 2.1 the GPU has a global device memory separated from the host system. The global memory is the most abundant and all stream cores on the GPU may access it. In order for the GPU's stream cores to access global memory it must first be copied from the host to the GPU device over the memory bus. Different programming models handle host to GPU memory movement differently, but usually the memory copy must be explicitly programmed by the programmer. The global device memory is the memory with the

highest latency and the programmer should therefore strive to minimize the number of accesses to it.

The GPU also has both software managed on-chip memory as well as hardware managed on-chip memory in the form of caches. Some of the on-chip memory may be accessed by all stream cores in the same thread group and facilitates communication within the same thread group. The on-chip memory has far lower access latency than the global memory. In applications where several threads in the same thread group access the same memory, utilizing the on-chip memory can reduce global memory accesses and thus, reduce memory latencies.

In CUDA the L1 cache is partitioned into two parts where one of the parts are used as software managed on-chip memory. This is referred to by CUDA as *shared memory*. Shared memory allow the programmer to explicitly control (a part of) the L1 cache which is accessible by all threads in a thread group.

Newer CUDA cards also has its own read-only data cache on-chip. The read-only data cache is intended for data that is constant during the entire run-time of a kernel. As with all other on-chip caches the read-only cache can be read by all threads in a thread group.

Utilizing the on-chip memory can often lead to significant performance improvements, however, the on-chip memory is a scarce resource. If a thread group uses too much of the on-chip memory, this will block other thread groups from executing in parallel because there is not enough available on-chip memory for them. This reduces occupancy and can severely hurt performance. The programmer should experiment with the amount of on-chip memory used per thread group and the size of the thread group, in order to find the configuration that provides the highest performance. As with the size of a thread group, this implies a large development cost while developing optimized GPU code. The amount of on-chip memory may vary from GPU to GPU, as well as the number of parallel thread groups allowed per multiprocessor, meaning that the application must be re-optimized when switching out the GPU.

All of the above in combination makes GPU programming complex and time-consuming. One of the shared memory optimizations for stencil codes reported in [36] resulted in a kernel consisting of more than 300 lines of code, which was ten times as much as the unoptimized kernel.

Given that was one of the many possible optimizations available and that one often has to try several optimizations before finding the best one, GPU programming often becomes prohibitively expensive for our target-user. The development cost incurred by porting serial code to CUDA, experimenting with different optimizations and configurations is something our project aim to decrease. As a side effect, our project may also help decrease the added development cost inflicted by making changes to the hardware.

2.1.4 Many Integrated Core Architecture

Intel's[1] MIC architecture[27] is quite new and the architecture comes as a new accelerator/co-processor. The current product line is named Xeon Phi. It is a standalone card connected to the host-system through the PCIe bus, like the GPU is. Second generation Xeon Phi cards can either be controlled by the host-system or it can act as the host-system itself. On the Xeon Phi card there are a high number of CPU cores, compared to a traditional CPU, but far fewer cores than on a GPU. The cores used on the Xeon Phi are x86 compatible and based on the cores used in the Intel Pentium processor. The x86 compatibility allows for use of existing parallelization software tools. Examples of existing tools that may be used is OpenMP, Open Computing Language (OpenCL), specialized versions of Intel's Fortran, C++ and math libraries[27, 29].

The cores in the MIC architecture are very similar to the multicore CPU cores and this might be of benefit when it comes to writing applications for it. Contrary to GPU programming, programming a MIC might be much more similar to programming a multicore CPU. The MIC's memory architecture is quite different from the GPU. Each core in the MIC architecture has its own L1 cache and the access latency for the cache is one cycle. The cores on the card are connected via a 512-bit bi-directional ring bus. The MIC contains its own separated RAM, akin to the GPU.

The card supports 512-bit SIMD instructions. The SIMD support together with the high number of cores with hyper-threading support lets the architecture compete with the GPUs. Hyper-threading is Intel's proprietary simultaneous multithreading (SMT) technology that allows several virtual (logical) threads to be run on the same physical processor

core, effectively increasing parallelism[26].

Intel's Xeon Phi cards offer both double- and single-precision performance that is comparable to, and in some cases better than GPU performance[28].

2.2 Application Characteristics

In order to narrow down our problem area, we have focused on applications that exhibit some specific traits. In this thesis we will focus on one of the "seven dwarfs"[5] both with respects to what our chosen programming model accepts in terms of application characteristics and test codes used while evaluating our results (see chapter 6). However, the programming model and source-to-source translator presented in this thesis may be used for other applications as well, although our translator might not perform optimal for them. The applications we focus on are stencil computations performed on regular Cartesian grids in three dimensions that exhibit high levels of parallelism. Although, it is worth noting that our translator works with one- and two-dimensional computations, as well. A well-known property of stencil computations is that they are often "memory bound", meaning that it is the machines memory bandwidth that limits performance[66]. In this thesis we will therefore focus on memory optimizations. Example applications of stencil computations include physical simulations like seismic wave propagation, fluid dynamics and heat propagation. Stencil computations also have uses in multimedia applications and are used for image smoothing (lowering contrast) and computer vision, to name a few.

Stencil computations involve computing a given point based on the point itself and a number of its neighbours. The number of neighbours and which neighbours included in the computations are dependent of the type of stencil. Stencils with a higher number of neighbours provide higher precision than a stencil with fewer neighbours included in the computation. A 7-point 3D stencil and a 19-point 3D stencil are illustrated in figure 2.2. The figure illustrates which points surrounding the center point that are included in the computation for the given center point. The following equation shows the computation of a three-dimensional 7-point Laplace operator that we will use as a running example illustrating the

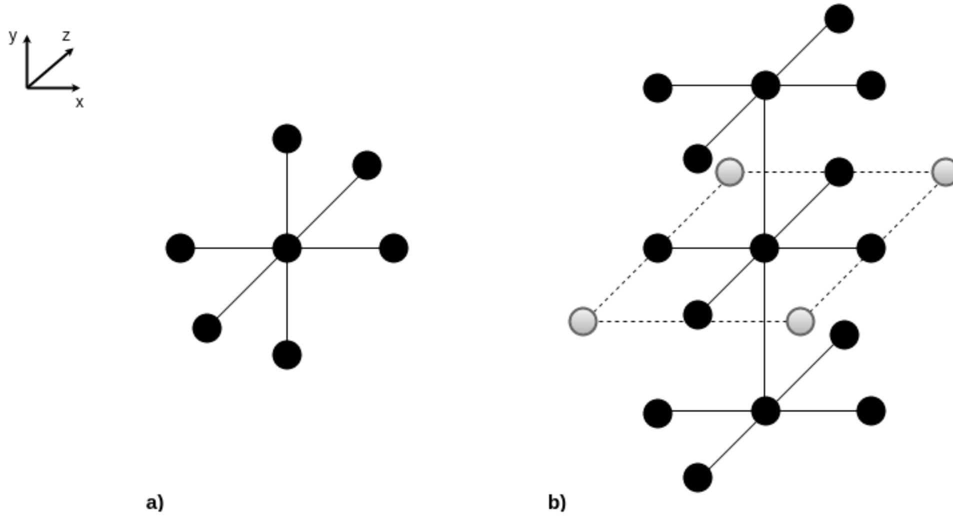

Figure 2.2: Two example stencils. a) shows a 7-point 3D stencil and b) shows a 19-point 3D stencil.

steps in our source-to-source translator later in this thesis.

$$u^{n+1}_{i,j,k} = c_1 * u^n_{i,j,k} + c_2 * (u^n_{i,j,k-1} + u^n_{i,j,k+1} + u^n_{i,j-1,k} + u^n_{i,j+1,k} + u^n_{i-1,j,k} + u^n_{i+1,j,k})$$

In the equation n denotes the time-step/iteration of the computation. The variable u denotes the Cartesian grid and c_1 and c_2 are constant coefficients. The subscripts i, j and k represents the indexes in the x, y and z dimensions, respectively. Note that the access pattern of the stencil is *strided* or *uncoalesced*, this means that the memory accessed does not lie contiguously in the Cartesian grid, or in programming terms, array. For some computing architectures this can degrade performance in that uncoalesced memory accesses are slower than coalesced ones. The reason for that is that some computing architectures, e.g Nvidia's CUDA GPUs, reads and caches more than one value at a time, in anticipation of an access to the contiguous values[14, section 5.3.2].

We focus our research on stencil implementations that use distinct arrays for reading from and for writing the final computation to. These arrays are swapped for each iteration/time-step of the computation. The fact that some of the arrays are only for reading, and the others are only for writing dramatically simplifies the parallelization of the computation. Using several arrays avoids the race conditions that arise in the single-array implementation. The computations selected for parallelization must

therefore allow for computations in an arbitrary order for each point.

It is typical for stencil computations that they have very few computations per memory access, also referred to as a high byte-to-flop ratio. This feature often makes stencil computations memory bound and optimizing memory accesses is an optimization with the highest potential. In stencil computations, neighbouring points have overlapping memory accesses as the computation of a central point include several neighbouring points. The neighbouring points in one computation is in themselves a central point for a different spacial index, as well as being a neighbouring point for several other central points. As an example, the 7-pt three-dimensional Laplace operator discussed earlier reads seven points in order to perform its computation for a given center point. All of the read points are being used in the computation of the neighbouring points in the different dimensions, with the exception of the edge points of the problem size. Sharing the memory accessed by a point with its neighbouring points in an efficient way to reduce processor time spent waiting for memory accesses often leads to performance improvements.

2.3 Parallel Programming Models

The traditional programming languages were designed for serial execution on a CPU. With the advent of GPGPU and the radically different architecture of the accelerators, there is a high demand for good languages and tools for developing applications that utilize accelerators fully.

In order to realize the performance potential of these accelerators, the inherent parallelism in the accelerators need to be utilized fully. The main task of the programmer is to expose the inherent parallelism of the application he/she is working on. In some cases, this might entail a larger redesign of the original code. The programmer must also handle memory movement between the host-system and the accelerator, as a result of the separated memory introduced by adding an accelerator to the system. Accelerators introduce a special on-chip memory-architecture. In an accelerators memory-architecture, different kinds of memory are often better for some purposes than others. It falls upon the programmer to choose which memory to use, in order to achieve the best performance. Performing the aforementioned tasks often require a significant programming effort and

can be quite time-consuming.

There has been a lot of effort put into creating parallel programming models. The proposed solutions often vary in the abstraction level they work on. Some solutions are quite low level, while others are at an even higher level than the programming languages they are extending. An example of a low level abstraction is the CUDA programming model. CUDA gives the programmer detailed control on the thread-level and access to intrinsic functions, for example. An example of a high level abstraction is OpenMP which we will discuss further in 2.3.2. OpenMP lets the programmer select what sections of code to run in parallel, but does not provide detailed control of how it is done.

The main benefit of a high-level abstraction is that it simplifies the programmer's view of the architecture. The programmer does not need to know low-level details of the hardware in order to write code that execute correctly on the accelerator. This leads to more rapid development. A drawback of the high-level abstraction is that the compiler looses detailed information about the programmer's intentions that it could have used to generate more optimized code. The programmer himself also looses the ability to perform low-level optimizations in the source code. These points summed up often leads to sub-optimal performance for programming models with high-level abstractions.

The low-level abstraction layer is good in the sense that it gives the programmer a higher degree of control and allows the programmer to perform lower-level optimizations. It also gives the compiler more occasions to perform optimizations, all in all producing more optimized code. However, low-level abstractions are more demanding of the programmer and his/hers knowledge of the underlying architecture. The level of detail generally decreases programmer productivity. If the perceived amount of work needed in order to parallelize an application seems too large, the task might be deemed too time-consuming and costly that it is dropped altogether. The complexity introduced by low-level abstractions may be seen as a drawback as it raises the threshold to utilize them.

In the following sections we describe Nvidia's CUDA programming model, which is used for programming Nvidia's GPUs. In subsection 2.3.2, we describe OpenMP. OpenMP is a parallel programming model

that historically has targeted multi-core CPUs. We then move to give a very brief description of the Open Accelerators (OpenACC) programming model. OpenACC's programming model is similar to OpenMP's programming model in that they both are annotation based. The main difference being that OpenACC specifically targets accelerators, while OpenMP only recently introduced accelerator support. Lastly, we describe OpenCL. OpenCL is the programming model that most closely resemble CUDA. The main difference between CUDA and OpenCL is that OpenCL targets a wide range of accelerators, while CUDA only targets CUDA-capable GPUs.

2.3.1 Nvidia's CUDA

CUDA is a general-purpose parallel-computing architecture and programming model created by Nvidia[65]. The platform facilitates performing GPGPU programming on a CUDA-capable GPU.

CUDA has a rich Application Programming Interface (API) and CUDA offers some alternatives in how to utilize the GPU. CUDA offers GPU-accelerated libraries that the programmer may use as a drop-in for already existing CPU-libraries. A programmer may also utilize the GPU through compiler directives, which we describe in 2.3.3. The last alternative is using CUDA through extensions to the C, C++ and Fortran languages. In addition to the language extensions there exists third-party wrappers to Python, Java, MATLAB, Perl, and others. Using CUDA through language extensions generally yield higher performance, or at least provides the programmer with the flexibility to implement optimizations that lead to higher performance.

The CUDA programming constructs map directly onto Nvidia's CUDA-capable GPUs and their memory model and threading hierarchy. CUDA-capable GPUs maintain their own separate memory on-chip where some of it is software managed, and some of it is hardware managed. The separated memory space affects the programming model, and CUDA provides mechanisms to allocate, copy and free memory for the different kinds of memory.

CUDA has a notion of *kernels*. In CUDA a kernel is essentially a program function that is run on the GPU, with some differences to the

C/C++ syntax. A CUDA kernel contains the code that all threads execute. A kernel is executed by a number of threads and these threads are divided into thread blocks (referred to as thread groups in subsection 2.1.3). Upon kernel execution the number of thread blocks and the number of threads per thread block are specified programmatically. Figure 2.3 shows an overview of CUDA's thread and memory hierarchy.

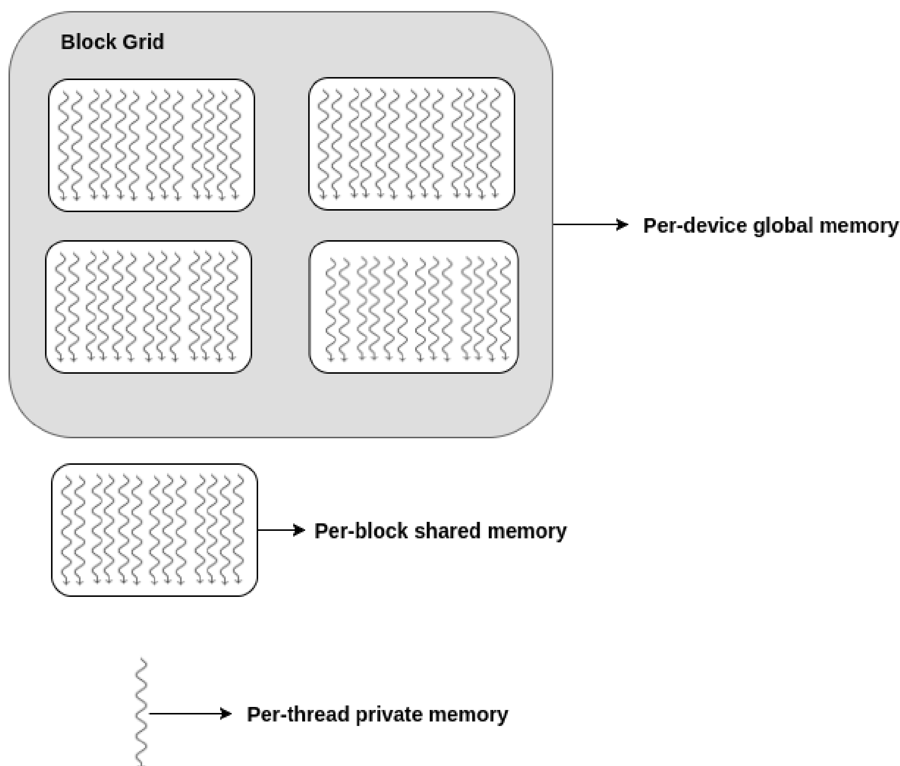

Figure 2.3: Thread and memory hierarchy in CUDA.

On CUDA GPUs thread-blocks are assigned by hardware to a Streaming Multiprocessor (SM) (or SMX, in the case of Kepler), which handles the execution of the threads in that thread block. Note that we referred to a SM as a multiprocessor in subsection 2.1.3. Thread blocks are logically executed concurrently on CUDA, but the programmer has no control over the order in which the thread blocks are executed so there should be no dependencies between them. CUDA does not allow for inter-block synchronization, only the threads within a thread block may be synchronized.

The threads in a thread block are further divided into *warps*. All threads in a warp should preferably perform the same computations and contain

no branch divergence. The reason for this is that threads in a warp are executed in a SIMT fashion. In cases where some threads in a warp are divergent, it will lead to serial execution of the divergent threads, lowering parallelism[13, section 12.1]. Currently, CUDA allows a maximum of 32 threads in a warp, and the size of the thread blocks should be a multiple of 32 in order to maximize utilization of the warp schedulers.

CUDA GPUs contain several different kinds of memory and a CUDA programmer should take care to utilize the memory best suited for the different variables inside a kernel. Memory-bound kernels can improve performance significantly by optimizing its memory usage.

The memory hierarchy consists of global memory and texture memory (constant) that is accessible by all threads, shared memory that is accessible for all threads in a thread block and registers that are per-thread. The global memory is the most plentiful memory, but it has the slowest access latency as it is placed off-chip. Shared memory is scarcer and on-chip per SM, meaning that all thread blocks running on a SM shares the resource. All threads within the same thread block may access the shared memory allocated for the thread block, allowing inter-thread synchronization within a thread block. Using too much shared memory per thread block result in lower occupancy, because there is not enough shared memory to have the maximum number of thread blocks per SM executing concurrently. However, shared memory has lower access latency than global memory and should be used in kernels with a high degree of inter-thread resource sharing to reduce global memory accesses. Finding the optimal balance between device occupancy and shared memory usage for a kernel often require experimentation with different configurations.

2.3.2 OpenMP

Open Multi Processing (OpenMP) is a directive-based programming model[7, 46]. OpenMP has support for C, C++ and Fortran. There are several compilers implementing the programming model, both commercial and open-source, and support for OpenMP exist for all major operating systems[45].

By adding annotations in the source code, the programmer guides

the compiler in its handling of the different parts of the code. These annotations may define loop nests that are to be parallelized, data copying and synchronization, to give a few examples[7]. Such user-directed parallelization, where the programmer explicitly specifies the actions made by the compiler, greatly simplifies the development of parallel programs. It also makes the code portable across platforms independent of the underlying hardware architecture.

The compiler bases its decision-making on annotations inserted by the programmer. For example, when the compiler detects a for-loop that is to be parallelized the compiler will generate code that creates several CPU-threads that shares the work in the loop. Historically, OpenMP has been used for multi-core processors but from OpenMP version 4.0 there is support for accelerators, such as a GPU.

2.3.3 OpenACC

Open Accelerators (OpenACC) is a programming standard for parallel computing[43]. The programming standard is directive based, meaning that the programmer annotates his/her code using OpenACC directives, much like OpenMP does. The languages supported are C, C++ and Fortran. OpenACC intends to simplify the programming effort required to program heterogeneous CPU/GPU systems.

OpenACC has a quite extensive collection of directives and clauses that may be used in guiding compilation. These directives give the programmer control of memory movement, execution configuration, synchronization, and more[44].

2.3.4 OpenCL

Open Computing Language (OpenCL) is a programming language that targets heterogeneous systems consisting of a traditional CPU and an accelerator. The accelerator may be a GPU, a Field-Programmable Gate array (FPGA) and other processors or hardware accelerators may be used. The language is an extension to C, and provides functionality to control and execute programs that utilizes the entire system. OpenCL provide both task-based and data-based parallelism.

OpenCL is an open standard that is maintained by the non-profit organization Khronos Group[58]. Conforming accelerator vendors include AMD, Intel, Nvidia, ARM and Apple, to name a few.

OpenCL code is hardware independent, meaning the same code will run on several different accelerators, which is an obvious advantage. However, in order to achieve the most optimized result, modifications often need to be made when working with different hardware architectures[30]. This is a result of the gap in the mapping between the programming model and the underlying hardware. OpenCL is more general in order to be applicable to a wider array of accelerators, and this makes the compiler's task more difficult. The paper in [30] also showed that CUDA outperformed OpenCL, and that additional manual alterations, like loop unrolling, applied to the OpenCL code was necessary to improve performance. In [20] CUDA were shown to outperform OpenCL on CUDA GPUs by as much as 30% in some cases. The performance gap between CUDA and OpenCL will most likely decrease as OpenCL matures and OpenCL-conformant compilers gets better.

2.4 Summary

In this chapter we discussed the current trends in computing architecture and that we believe that the massively parallel trend will continue. This trend is caused by an increasing difficulty to produce traditional single-core processors that are able to deliver comparable performance to multicore processors or accelerators. We provide a discussion on the effects the current trend of computing architecture has on the complexities of application development. In section 2.2, we provided a description of the type of applications we focus on in this thesis and what they are used for. Lastly, in section 2.3, we introduce some of the most widely used parallel programming standards with a brief discussion about the pros and cons of low- and high-level abstractions. The current state of computing architectures and the common tools used to program them often lead to a situation where one have to choose performance over ease of programmability or vice versa, and that is the motivation for this thesis. We would like to offer both ease of programmability and high performance and we hope to achieve this with our source-to-source

translator that accepts Mint-annotated C and C++ code. In chapter 3 we describe the Mint programming model, the Mint model's annotations to the C/C++ languages and a Mint-conforming source-to-source translator.

Chapter 3

The Mint Programming Model

In this chapter we present the Mint programming model. We start by providing some background information. Then we will move on to explain the model on a higher, more general, level. We will describe the Mint model's assumptions about the hardware system it is intended to work on. We then discuss Mint's execution and memory model, and the restrictions Mint imposes on the programmer. In section 3.2, we provide a detailed description of the Mint interface in the C and C++ languages. Lastly, we describe a source-to-source translator that implements the Mint programming Model.

3.1 The Model

The Mint model was developed by Didem Unat in her PhD thesis[61]. The Mint model is a high-level abstraction so that the non-expert programmer may guide a Mint-conforming translator in its process of generating optimized source code that utilizes an accelerator for the given input code. The model was designed especially for stencil computations, but the programming model may be used for other computations that have a high degree of parallelism as well. The Mint model's two major goals were to 1) increase the productivity of the programmer, and 2) provide performance comparable to hand-optimized CUDA code[61]. In addition to these two major goals the model provides the programmer with a way to incrementally optimize and test code, without in-depth knowledge of the hardware. The incremental development of the

Mint Model interlocks nicely with CUDA's Assess Parallelize Optimize Deploy (APOD) development-cycle[13]. Additionally, a Mint-conforming translator that generates CUDA code can be used as a learning tool for the novice CUDA programmer by inspecting the generated code.

The programming model assumes a traditional multi-core system with a massively parallel single-chip connected through a memory-bus, e.g. PCIe[48, 61]. Further in this thesis we will refer to the traditional multi-core system as the *host*, and the massively parallel single-chip as the *accelerator*, or simply *device*.

Assumptions the Mint model makes about the accelerator includes that it has its own memory, separated from the rest of the system. Explicit memory-transfers between the host and the accelerator via the memory-bus are therefore needed. A CUDA capable GPU is an example of an accelerator that fits Mint's assumptions[13, section 2.1].

A Mint conforming program consists of Mint-annotated C/C++ code[61, p. 27]. Prior to the master project presented in this thesis, only Mint-annotated C code was supported by existing translators. Our project expands this to include C++ as well.

Mint annotations are compiler directives inserted into the C/C++ source code and the Mint-conforming translator uses these directives in its decision-making during translation. In section 2.3.2 we discussed OpenMP. The Mint directives bear some similarities with OpenMP directives, but they are far fewer, simpler and Mint include a special purpose directive for data movement. The Mint directives informs/guides the translator, currently there are no executable directives in Mint[61, p. 27]. Mint currently consists of the following 5 directives:

1. *parallel* - Defines a parallel region. A parallel region may contain one or more accelerated regions (see pt. 2) to be optimized by the translator.

2. *for* - Defines a loop-nest to be parallelized, optimized and run on the accelerator. In this thesis we refer to such a loop-nest as an *accelerated region*. The directive must be inside a Mint-parallel region.

3. *copy* - Instructs the translator to generate code that moves data to or from the accelerator.

4. *barrier* - Informs the translator to explicitly synchronize the host and device.

5. *single* - Defines a region inside an accelerated region to be executed by a single thread.

3.1.1 Mint's Execution Model

The Mint model assumes that a block of parallelizable code may be executed on the accelerator and that it is the host who initiates and guides this execution[61, p. 28]. A valid Mint program contains one or more parallel-regions and inside of these regions there may exist one or more parallelizable loop-nests to be executed on the accelerator. It is the programmer's job to identify computationally-heavy loop-nests that may be accelerated. When the programmer has identified candidate loop-nests, the programmer informs Mint of which loop-nests to parallelize by inserting a Mint *for* directive before the outermost-loop of the loop-nest. Mint takes the loop-nest following a *for* directive and generates accelerator-specific code that performs the same computations on the accelerator.

In addition to generating the accelerator-specific code, Mint also generates host-side code that performs any pre-execution configurations required and code that executes the generated accelerator code. Mint explicitly synchronizes the host and the device after the execution on the accelerator. This synchronization may be omitted by using the *for* directives *nowait* clause. The programmer may then explicitly synchronize the host and device at a later point in the host-code by inserting a Mint *barrier* directive. We will describe the directives and their clauses in greater detail in section 3.2.

3.1.2 Mint's Memory Model

The Mint model assumes that the accelerator is connected to the host-system via a memory-bus and that it maintains its own separate memory space. Memory that is accessed inside a Mint *parallel* directive needs to be transferred to the device in order for the device to perform its computations. The Mint model performs memory allocation on the

accelerator, memory movement between the host and the accelerator (in both directions) and memory deallocation of the accelerator's memory. However, Mint requires the programmer to guide the translator with respects to what memory to move to or from the accelerator, and the size of that memory. The programmer performs this guidance through the insertion of Mint *copy* directives before, and after the entrance of a Mint *parallel* region. It is therefore important that the programmer is aware of the separate address spaces of the host-system and the accelerator. In order for the host to access the results of a computation on the accelerator, the programmer must insert *copy* directive(s) at the exit of a Mint *parallel* region. At the exit of a *parallel* region and after any memory copies back to the host, Mint-conforming translators deallocates all device memory allocated for that *parallel* region. The contents of the device memory will then be lost. Because of the deallocation of device memory on the exit of a parallel region there can be no communication between separate parallel regions through device memory[61, p. 28].

The reason that the memory movement need to be guided by the programmer is because it is a hard problem to automate reasoning about dynamic memory allocations. The task is further complicated by C/C++'s pointer-aliasing functionality, and the Mint model therefore require the programmer to guide the translator.

The Mint model does not give the programmer direct access to the accelerator's memory. The model manages this memory independently and this is part of the Mint model's abstraction. The programmer does not need knowledge about the different kinds of on-chip memory and caches. However, the programmer may guide the Mint translator's handling of on-chip memory through the use of compiler options. This way the programmer may experiment with different optimizations without having an in-depth knowledge of the underlying hardware. Any knowledge the programmer already possess about the underlying architecture may be used to decrease the optimization search-space, when experimenting with different configurations.

3.2 The Mint Interface

The Mint directives are represented in the source-code by inserting pre-processor directives or "pragmas" through the #pragma mechanism of both C and C++. A Mint directive always starts with "#pragma mint" followed by the chosen Mint directive. The syntax is identical across both languages, simplifying both working with Mint-annotated code and implementing a Mint-conforming translator. A regular C/C++ compiler that does not support Mint will simply ignore the Mint directives. Adding Mint directives to existing code is therefore harmless in the sense that it will not break it. Listing 5.1 in section 5.2 shows an example of a Mint annotated C++ program. In the following subsections we will describe the Mint directives and their clauses in further detail.

3.2.1 The Parallel Directive

The parallel directive defines a parallel region. The syntax for a parallel region is:

#pragma mint parallel

The parallel directive must be followed by a compound statement ({ . . . }), on the following line. We will hereby refer to a compound statement following a parallel directive as a parallel region. A parallel region envelopes all code that is to be accelerated on the device, and optionally, code that is run on the host. A parallel region can not span multiple source files or functions/methods. Nested parallel regions, e.g. a parallel region within a parallel region, is not allowed in Mint. Additionally, Mint does not allow branching out of a parallel region.

Further in this thesis we will refer to all dynamically allocated arrays defined outside of the parallel region and referenced inside the parallel region as *externally declared arrays*. All externally declared arrays must be explicitly copied to the device through the Mint *copy* directive directly before the entrance to the parallel region. Otherwise the translator will print an informative error-message and exit. If device-memory needs to be accessed by the host after the completion of a parallel region, the memory must be explicitly copied back to the host through the Mint *copy* directive directly after the parallel region.

3.2.2 The For Directive

The Mint *for* directive marks the following for-loop as a candidate for execution on the accelerator. We will hereby refer to a for-loop annotated with the for directive as an accelerated region. Its syntax is:

#pragma mint for [optional clauses]

The directive must be placed inside a parallel region, and as with the parallel region, there can be no branching out of the accelerated region. The for directive may be followed by optional clauses and a for-loop (optionally nested) is required on the next line. The loop-nest to be parallelized may not contain any expressions or statements in-between the for-loops.

The Mint model does not allow nested for directives, e.g. a for directive inside the loop nest of another for directive. However, newer CUDA capable GPUs (compute capability >= 3.5) supports a feature Nvidia calls *dynamic parallelism*[14, Appendix C]. Dynamic parallelism allows a thread on a CUDA GPU to spawn additional child threads. Nesting parallel work on CUDA accelerators is shown to be beneficial in several applications, as it may reduce memory bus occupancy, achieving higher launch throughput and reducing false dependencies[3]. Researching the possibility of offering nested for directives in the Mint model in order to utilize such a feature remains as future work.

The following optional clauses to the Mint for directive may be used to configure the execution on the accelerator:

- **nest(*all* | #)**
  The nest clause allows the programmer to specify how many of the nested for-loops are to be parallelized. Either the keyword *all* or a positive integer ranging from 1-3 may be used. If the clause is omitted, the nesting level defaults to 1. Of the for-loops to be parallelized Mint always assigns the innermost one as working in the x-dimension (fastest varying dimension), the next-innermost one as the y-dimension and the outermost one (given a nesting level of 3) as the z-dimension.

- **tile(N_x, N_y, N_z)**
  The tile clause allows the programmer to specify the dimensions of

a tile, which contains the items to be computed by one thread-block. A tile is the most coarse-grained granularity Mint provides. The clause accepts between one and three positive integers, separated by a comma. The first integer specifies the number of items in the x-dimension, the second integer (if any) assigns the number of items in the y-dimension and the third integer the number of items in the z-dimension.

If the tile clause is omitted, the source-to-source translator presented in this thesis uses default values of 16, 16 and 4 in the x, y and z dimensions, respectively. The default values are configurable.

- **chunksize(N_x, N_y, N_z)**
  The chunksize clause allows the programmer to specify the number of elements each accelerator thread computes in each dimension. The clauses syntax is the same as for the tile clause. The number of threads per thread block is determined by the tile size in a dimension, divided by the chunksize of the corresponding dimension. Example, $Tile_x/Chunk_x = Thread_x$ where $Tile_x$ is the tile size in the x-dimension, $Chunk_x$ is the chunk size in the x-dimension and $Thread_x$ is the computed number of threads in the x-dimension. The optimal number of items per thread often depends on the accelerator architecture, the computation performed and the optimizations used. By using these clauses the programmer can easily experiment with different configurations in order to find the best configuration. If the chunksize clause is omitted, our source-to-source translator use a default value of 1 in all dimensions.

- **nowait**
  The nowait clause may be used to instruct the translator to omit generating code that explicitly synchronizes the host and device after the execution of an accelerated region on the accelerator.

  This clause may be useful when working with a translator that generates CUDA code, for instance. CUDA has a concept of streams, where one stream is a queue of memory transfers and kernel executions[13, section 9.1.2]. CUDA executes work from a stream in order, effectively providing implicit synchronization for work added to a given stream. Mint places all generated memory

transfers and kernel executions on the same stream. Generated code that adds several kernel executions after one another without the need of accessing device-memory on the host in between, may therefore benefit from using the nowait clause, as it reduces explicit synchronization between the host and device. The host may then queue up all the work, and delay explicit host-device synchronization to the point where the host need to access device-memory. This reduces the number of host-side CPU interruptions, provides a contiguous stream of work to the accelerator and frees up the CPU to perform other tasks while it waits for the accelerator to perform its work.

- **reduction(operator:var)**
  The reduction clause may be used to inform the translator that the reduction operator specified in "operator" is to be applied to the scalar variable indicated in "var" on the accelerator. Note that this clause is not implemented in Spearmint, and only partially in the Mint translator[61, section 3.5.5].

3.2.3 The Copy Directive

The Mint programming model requires the programmer's guidance in handling memory movements. Mint's copy directive is the programmer's interface to inform Mint of which memory to move to and from the accelerator, and the dimensions of said memory. The copy directive's syntax is as follows:

#pragma mint copy(src | dst,*toDevice* | *fromDevice*,N_x,N_y,N_z)

The *src* and *dst* identify the pointer to the host-side memory to transfer, and they need to be declared before the copy directive. Mint will generate new pointers to the device-side memory and keep track of the mapping between them. Mint will allocate device-side memory, move the data over the memory-bus to the device (or to the host) and deallocate the memory after use, automatically. The *toDevice* and *fromDevice* keywords specifies if the memory copy is from the host to the device, or from the device to the host, respectively. Following the direction keyword, between one and three expressions are defined, each of these must evaluate to a positive

integer. The expression(s) determines the dimensionality of the memory to be copied, where the first (leftmost) expression specifies the fastest varying dimension (x-dimension), and the following expressions specifies the y- and z-dimensions, respectively. Mint use the expressions while generating code that transfers data between the host and the accelerator. If these expressions do not match the actual size of the host-side memory specified, it will lead to memory errors and undefined behaviour. A Mint copy directive with the keyword *toDevice* must be placed directly before a parallel directive, without any statements in between them in order to copy memory to the device. Similarly, if the *fromDevice* keyword is used, the directive must directly follow a parallel region without any statements in between.

3.2.4 The Barrier Directive

The Mint *barrier* directive's syntax is as follows:

```
#pragma mint barrier
```

The Mint barrier directive should be used when there is a need for explicit synchronization between the host and the device. For instance, CUDA's kernel-calls are executed asynchronously from the host, and this is something the programmer may use to his/her advantage when searching for the most optimized Mint configuration[14, Appendix B.1.2]. A situation that will require the use of the barrier directive will only arise in code that utilizes the *for* directives *nowait* clause. If the nowait clause is omitted, a Mint-conforming translator will generate code that explicitly perform host-device synchronization after running computations on the accelerator. Errors due to synchronization errors will never occur in such a case.

3.2.5 The Single Directive

Mint's *single* directive defines a single block where the code in the single block is executed by exactly one accelerator thread. The single directive's syntax is:

```
#pragma mint single
```

The pragma must be followed by a compound statement ({ ...}) on the next line whose contents will be computed sequentially by a single accelerator thread. The single directive should be used sparingly as it dramatically decreases the parallelism on the accelerator. This directive is intended to be used in cases where it is beneficial for performance to perform typically host-side actions on the accelerator, in order to avoid memory transfers between the host and the device. Note that the source-to-source translator presented in this paper does not yet support this directive.

3.3 The Mint Translator

In addition to developing the Mint model, Didem Unat also implemented a Mint-conforming source-to-source translator in her PhD work, named Mint[61–63]. To avoid confusion we will refer to it as the *Mint translator*, or simply the *translator* and to the Mint model, as the *Mint model*. The translator accepts Mint-annotated C code and generates CUDA C code.

The translator implements several optimizations. Mint's major optimization utilize CUDA's shared memory[13, section 9.2.2] which we described in subsection 2.1.3. The Mint translator requires that the chunking (set with the Mint model's *chunksize* clause) in the z-dimension is equal to the tile size (set with the Mint model's *tile* clause) in the z-dimension, while using Mint's shared memory optimization. This restriction effectively only allows two-dimensional thread blocks while Mint's shared memory optimization is used. The restriction allows reuse of memory loaded from global memory and placed in shared memory by rotating the z-planes in the shared memory, as illustrated in figure 3.1. The outermost z-planes contains the stencil's center point's neighbors in the z-dimension. The z-plane in the middle contains the stencil's center point, and its neighbors in the x and y dimensions. When utilizing CUDA's shared memory, Mint makes each thread load its central point into the shared memory space, followed by a call to CUDA's synchronization function, __syncthreads()[14, Appendix B.6]. The call to __syncthreads() is necessary to ensure that all values have been loaded into shared memory before they are read by any other thread in the thread-block. The synchronization incurs a small performance cost as all threads in the thread block waits until

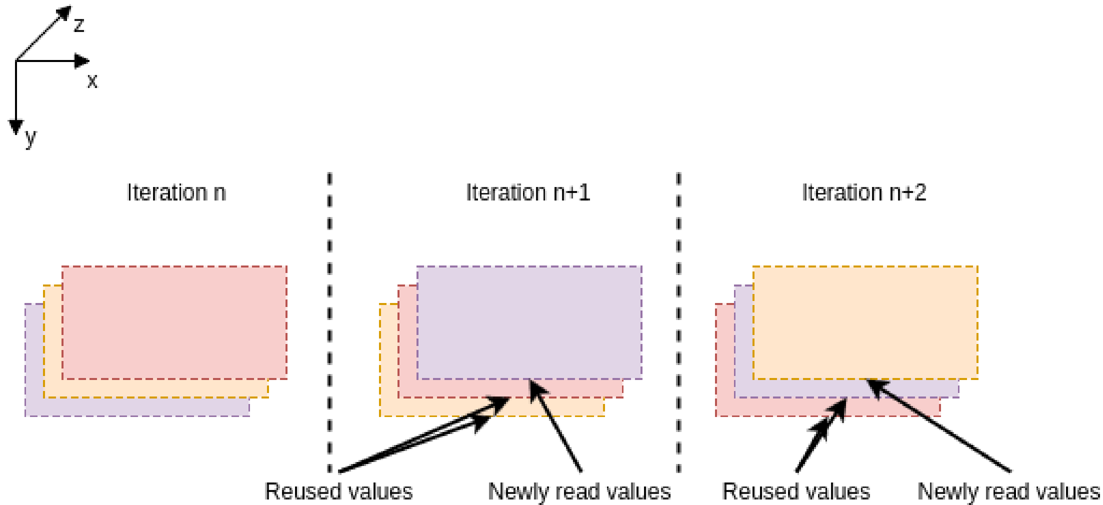

Figure 3.1: The Mint translator's reuse of data loaded into shared memory while chunking in the z-dimension. The values in a z-plane is reused for each iteration.

all memory is loaded from global memory. However, synchronization is necessary to avoid read-before-write and ensure correctness. The optimization also adds some branching (if-statements) in order to correctly load boarder points into shared memory. Branching on CUDA GPUs can hurt performance, but is needed for correctness[13, section 12.1].

The Mint translator also offer register optimization[61, section 5.6]. The register optimization places intermediate values in a thread's register instead of writing it to the L1 cache (if any) or L2 cache for values that are accessed several times by the same thread. This optimization uses more register space, which can lead to register spilling or lower device occupancy caused by shortage of register space, but can in return reduce L1- and L2-cache accesses and global memory accesses upon a cache-miss.

When both the shared memory and register optimization is used in combination and the stencil pattern allows for it, a further improvement is added. For stencils that only use one neighboring point in both directions of the z-dimension, such as stencil a) in figure 2.2, the translator places these two neighboring points in registers and only allocates the center plane illustrated in figure 3.1. This reduces the shared memory used per thread block by two-thirds.

Additionally some minor optimizations relating to avoidance of repeating the same computations several times in the generated code were

implemented. The generated CUDA C code (with optimizations) was shown to be comparable with hand-optimized CUDA code for some of the most common stencil computations[61, section 6.2.1]. In [63], the Mint model and the Mint translator was used to accelerate a 3D finite-difference earthquake simulation. The Mint-generated code that ran on a single Tesla C2050 card outperformed a multicore cluster named *Triton* running 32 Nehalem dual-socket, quad-core Intel Xeon E5530 processors using Message Passing Interface (MPI).

The translator's optimizations were developed with CUDA's Fermi architecture[41] in mind, as it was the latest architecture at the time of the translator's development. The optimizations performed by the translator do not necessarily perform as good on newer CUDA architectures, and that is some of the motivation behind this master project.

The translator utilizes the Rose compiler architecture[51], which requires older versions of common compilers like *GCC*[21] to build and install. Additionally, users have reported that installing the translator, which also requires installation of Rose, is cumbersome. As the Mint model's target-user ranges from the non-expert programmer, difficulty installing the translator is not ideal.

As noted earlier, the translator only compiles Mint-annotated C code to CUDA C, and there is a wish to expand this to include C++ as well.

3.4 Summary

In this chapter we have provided a detailed description of the Mint programming model. The Mint programming model is intended to allow novice programmers to easily utilize the power of a massively-parallel single-chip accelerator. In this chapter we have described how the Mint model facilitates this. We have described the models assumptions about the target-systems architecture and memory model. We have described Mint's execution and memory model that are designed to map easily onto the target-system's architecture. The abstraction Mint provides allows the non-expert programmer to focus on the problem at hand without detailed knowledge of the underlying hardware.

In section 3.2, we provide a detailed description of the Mint interface to the C and C++ languages, and the restrictions the interface place on

Directives and optional clauses	Description
parallel { . . . }	Defines a parallel region which may contain one or more for directives. Note the newline before the parallel region.
for nest(*all* | #) tile(N_x,N_y,N_z) chunksize(N_x,N_y,N_z) nowait	Defines a loop-nest to be accelerated. Defines nesting level. Defines the dimensions of a tile. Defines how many items computed per thread. Omits explicit host-device synchronization after running computation on the accelerator.
copy(dst | src, toDevice | fromDevice, N_x, N_y, N_z)	Moves data to or from the device.
barrier	Generates explicit synchronization
single { . . . }	Defines a region that is executed by only one thread. Note the newline before the single-block.

Table 3.1: Summary of the Mint directives.

the programmer. Table 3.1 shows a summary of the Mint directives, their clauses and their syntax in the C and C++ languages.

We also describe the Mint source-to-source translator that implements the Mint programming model in section 3.3. The Mint translator accepts Mint-annotated C code and produces optimized CUDA C code that performs comparable to hand-optimized CUDA code on Fermi devices. However, the Mint translator is difficult to install, it only supports C and its optimizations are intended for an older CUDA architecture. Creating a source-to-source translator that is easy to install and generate optimized code for the Kepler architecture in both C and C++ are three of my master project's major goals.

Chapter 4

The LLVM Compiler Infrastructure

In this project we wanted to build a Mint-conforming source-to-source translator using another compiler framework than Rose, which the Mint translator is based on. We have built our source-to-source translator with support from the LLVM compiler infrastructure. The LLVM compiler infrastructure provide several libraries and tools that perform common tasks related to compilation. Didem Unat's Mint translator utilize the Rose compiler architecture[51], which provides functionality for lexing, parsing, building Abstract Syntax Tree (AST), and more[61]. However, users have reported difficulties installing Rose. Additionally, using Rose has effects on the Mint translator's code generation, which is discussed in section 6.6. We chose the LLVM compiler infrastructure for several reasons and we will document these and give an overview of the infrastructure in this chapter. In section 4.1 we provide background information and a discussion of LLVM's design. In section 4.2 we will describe Clang and its libraries and tools. We will provide a more detailed description on some of Clang's libraries that are of special interest in our project.

4.1 LLVM

The LLVM compiler infrastructure has its origin from the University of Illinois where it was originally developed to provide a modern compiler capable of supporting both static and dynamic compilation of arbitrary

programming languages[23]. The LLVM project started in year 2000, under the leadership of Chris Lattner and Vikram Adve. The name “LLVM” was originally an acronym for *low level virtual machine*, but as the project scope grew the acronym was dropped due to confusion, as the LLVM project had little to do with traditional virtual machines. “LLVM” is now the name of the LLVM umbrella project[32, 33].

LLVM is open-source and licenced under the “UIUC” BSD-style license[23]. LLVM is cross-platform and known to work on Windows, Mac, Linux, Solaris, FreeBSD and others[22]. It is implemented in C++, and uses CMake[8] to generate native build files that defines how the project should be built. CMake may be used in conjunction with Unix MakeFiles, Visual Studio or Ninja, to name a few of the available build-systems.

The LLVM compiler infrastructure is a collection of modular, interchangeable and reusable compiler and tool-chain technologies. Examples of such tools are; LLDB (debugger), LLVM C++ standard library, LLVM's Clang (more on Clang in section 4.2) and LLVM's machine code sub-project to name a few[23].

Under the LLVM umbrella project there are several sub-projects. Many of these sub-projects are being used in production of both open-source and commercial projects, either as-is or new tools are being based on them. Examples of such projects are Nvidia's *nvcc* compiler, which is based on LLVM[15] and Apple's *Swift* language compiler which uses LLVM[54].

From its start in year 2000, LLVM was designed as a set of libraries with emphasis on well-defined interfaces. In 2012, the LLVM project received the ACM's software system award. It was awarded to Chris Lattner, Vikram Adve and Evan Cheng[2]. LLVM uses the traditional three-phase compiler design[32] consisting of a front-end, an optimizer and a back-end, as illustrated in figure 4.1. The aforementioned design coupled with

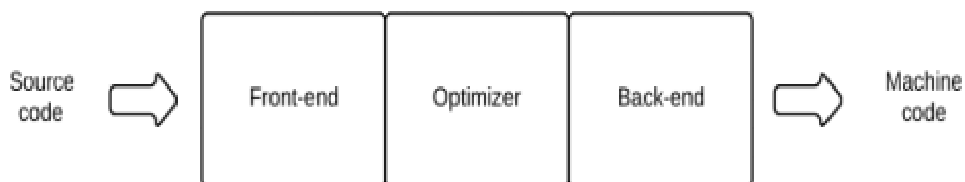

Figure 4.1: An example of a traditional three-phase design.

LLVM's Internal Representation (IR) is what makes LLVM modular. IR is a low-level RISC-like[49] virtual instruction set that resembles assembly. It is used as the LLVM's internal representation, and was designed to host mid-level optimizations and analyses. The IR uses a static single assignment form[1][18] for scalar register values, to aid in optimizing. You can see a small example of LLVM IR code in listing 4.1 which corresponds to the C code in listing 4.2.

```
1 define i32 @add(i32 %a, i32 %b) {
2 entry:
3   %tmp1 = add i32 %a, %b
4   ret i32 %tmp1
5 }
```

Listing 4.1: A simple example of LLVM's IR

LLVM's design makes it possible to clearly define the three-phases' tasks. The front-end performs pre-processing (in the case of C-family languages), lexical analysis, parsing and semantic checks and converts code in the source language to legal (doesn't have to be optimized in any way) IR-code. The optimizer receives the IR-code, optimizes it and sends the optimized IR-code to the back-end. The back-end then converts the IR-code to machine- or byte-code depending on the back-end's target architecture. The only communication needed between the

```
1 unsigned add(unsigned a, unsigned b) {
2   return a+b;
3 }
```

Listing 4.2: The corresponding code from listing 4.1 in C.

phases is the IR, which simplifies splitting up the workload and keeping the implementation of the different phases separate. The different phases and tools can then be chained together in many different permutations in order to achieve the desired output.

The IR is low-level, and can therefore be used to represent 'all' high-level languages similar to how you can map many source languages to microprocessors[35].

[1]Commonly abbreviated as *SSA form* or simply *SSA*.

An effect of this design is that when a LLVM front-end for a new language is created, there may already exist an optimizer and several back-ends for it. A new language *A*, may be compiled to target architectures *B*, *C* and *D*, if there exist LLVM back-ends for targets *B*, *C* and *D*. If there exist ten back-ends you may compile to any one of them, just by implementing the front-end for the new language. This is illustrated in figure 4.2. The only requirement for creating a front-end is that the input language may be represented by the IR.

This works the other way around as well. When a new LLVM back-end is created, every programming language with an already existing LLVM front-end can be compiled to the target of that back-end. The design also entails that an implementation of a new optimization in the optimizer-phase will impact all programming languages with a LLVM front-end and all machine-targets with a LLVM back-end. This results in a high degree of code reuse. The design dramatically lowers the development effort required to create a new compiler, as some, or most of the work likely is done before.

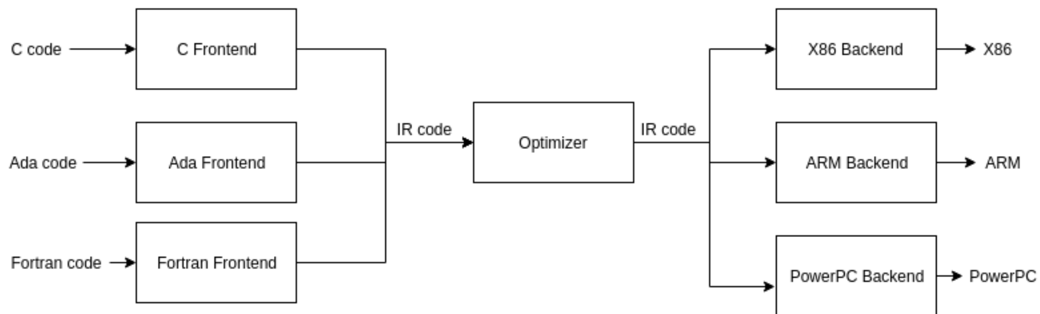

Figure 4.2: The modularity of the three-phase design due to LLVM's IR.

4.2 Clang

Clang[10] is one of LLVM's sub-projects and was created to be a LLVM front-end for several of the C-family languages. It supports C, C++, objective C and objective C++. It is designed to be GCC-compatible, and can be used as a drop-in replacement for GCC, simplifying the migration from GCC to Clang.

One of Clang's goals was to make it easy for developers with a basic understanding of compilers and working knowledge of C/C++

to contribute to the project. The Clang team have therefore kept the AST as close to the languages as possible. This was also some of the motivation behind making Clang's architecture made up of a collection of libraries. The library-based architecture allows new developers to contribute without knowledge of the full code-base. Only knowledge of the part they are working on is necessary. This is thought as lowering the threshold for new developers to contribute[11].

Clang is made up of the the libraries and tools seen in table 4.1[11]. We will utilize a subset of them in the source-to-source translator presented in this thesis.

libsupport Basic support library, from LLVM.

libsystem System abstraction library, from LLVM.

libbasic Diagnostics, SourceLocations, SourceBuffer abstraction, file system caching for input source files.

libast Provides classes to represent the C AST, the C type system, builtin functions, and various helpers for analyzing and manipulating the AST (visitors, pretty printers, etc).

liblex Lexing and preprocessing, identifier hash table, pragma handling, tokens, and macro expansion.

libparse Parsing. This library invokes coarse-grained 'Actions' provided by the client (e.g. libsema builds ASTs) but knows nothing about ASTs or other client-specific data structures.

libsema Semantic Analysis. This provides a set of parser actions to build a standardized AST for programs.

libcodegen Lower the AST to LLVM IR for optimization & code generation.

librewrite Editing of text buffers (important for code rewriting transformation, like refactoring).

libanalysis Static analysis support.

clang A driver program, client of the libraries at various levels.

Table 4.1: Clang's libraries and tools.

Clang provides a single unified hand-written recursive-descent parser

for the supported languages. It makes it easy to implement ad-hoc rules in C/C++ and makes it straight-forward to implement good diagnostics and error recovery[11]. We will discuss Clang's parser more in-depth in section 5.1.

One of the Clang project's major goals was to offer good diagnostics. Another goal was to allow easy integration with integrated development environments (IDE). Clang keeps a lot of source location information about the source code in the AST nodes, to help fulfil these goals.

Clang supports several frameworks like OpenMP, OpenCL and language extensions like Nvidia's CUDA[55](experimental as of Clang 3.8). The OpenMP support is helpful to us, as we want our source-to-source translator to allow both Mint directives and OpenMP directives to co-exist in the input source.

4.2.1 LibTooling

The Clang Team recommends using the LibTooling library to write source-to-source translation tools. LibTooling is a library that acts as an interface to standalone tools which makes use of Clang's libraries.

Several of the tools that come with Clang such as *Clang-format* (a tool that formats C++ code) and *Clang-check* (performs basic error checking and AST-dumping of source code) make use of the LibTooling library.

The library provides a common options parser (CommonOptionsParser) which is used to parse common command-line options such as input source files, compilation flags, etc. for Clang-tools. The CommonOptionsParser is built on LLVM's CommandLine[56] library which is built to be easily extended with a declarative approach. Command line argument values are transparently captured and can be accessed via user-defined variables. The job of error checking is largely left to the library and converting the command lines textual representation into for example integers or boolean values is done automatically by the library. The library allows its users to avoid writing boilerplate code, such as parsing the command line options yourself and printing help- and error-messages. This allows the tools using the library to focus on what they want to achieve. The CommonOptionsParser provides a consistent interface to all Clang-tools.

LibTooling provides a simple way to create a ClangTool. A ClangTool

can be used to run a FrontendAction over a translation unit[2]. ASTFrontendAction is an extension to the FrontendAction and may be used to run actions over the AST. ASTFrontendAction implements a method that returns an ASTConsumer that is an abstract interface to be implemented by clients that read the AST. ASTConsumer represents an abstraction layer that allows the client to be independent of the AST-producer. The ASTconsumer may be used to run your own RecursiveASTVisitor over a Clang-generated AST. The RecursiveASTVisitor performs pre-order depth-first traversal of the AST and visits every node. The framework creates visit functions for every node in the AST. For example for an AST-node called "Foo" there will be created a "VisitFoo" function. That function takes an argument of type "Foo", that is the visited node when the RecursiveASTVisitor calls the "VisitFoo" function. By extending the ASTConsumer, the Visit* functions may be overridden to provide custom handling of the AST-nodes of interest.

4.2.2 LibRewrite

Clang's rewriter library (LibRewrite) is a library that facilitates source-level transformations. We used LibRewrite extensively in our source-to-source translator and we therefore give a brief overview of its capabilities. In section 5.2 we describe just how we used LibRewrite in our translator.

LibRewrite is a high-level interface to the rewriter's buffers and dispatches requests to modify the lower-level buffers that contain the source-code in its original form. The library offers functionality to replace an arbitrary-length part of the source-code. It also offers functionality to insert text into the source-buffers. The library accepts a *SourceRange* object or a *SourceLocation* object that specifies the location to replace or insert text, respectively. LibRewrite accepts raw text to either replace existing code or to insert into the source-buffers. The text must be represented in a std::string object and there are no restrictions on the contents of the string. The inserted text may therefore be source-code, comments, defines, blank spaces, etc.

LibRewrite does not perform any lexical or semantic analysis on the modified source-buffers and it is therefore important that the modifica-

[2] A file containing source code.

tions performed are legal with respects to the source-language.

LibRewrite keeps track of all modifications and offsets all original source-locations appropriately. LibRewrite greatly simplifies modifying the original source as we do not need to worry about how earlier modifications have altered the source buffers and the AST-node's source-locations.

LibRewrite may be used in concert with Clang's LibTooling library. For example it may be used to create an ASTFrontendAction that visits selected nodes and modifies them at source-level.

4.3 Summary

In this chapter we have presented the LLVM compiler infrastructure. We have provided an overview of its origins and motivations and a discussion of its three-phase design. In section 4.2, we focused especially on Clang, one of LLVM's sub-projects. Clang is a C-family front-end for the LLVM compiler infrastructure built with emphasis on providing its different functionality as libraries. Clang's libraries provide functionality for lexing, parsing, AST-generation, code-generation and more. We have described some of Clangs libraries in greater detail as we will utilize them in our implementation of our source-to-source translator. Chapter 5 describes how our source-to-source translator is implemented.

Chapter 5

Spearmint Source-to-Source Translator

In this chapter we discuss our source-to-source translator, Spearmint. In chapter 4 we described LLVM, Clang and several of Clang's libraries. The libraries described provide a way to traverse the AST, and it is therefore important that all the information we are interested in gets included in the AST. This is not the case for Mint directives in out-of-the-box Clang. We therefore modified Clang, in order to represent Mint directives in the generated AST. In this chapter we start by briefly describe the modifications we made to Clang. We then provide a detailed description of our implementation of our source-to-source translator. We outline Spearmint's different parts and the tasks that the different parts perform.

We have implemented an optimization that utilize newer CUDA GPU's read-only cache in our translator. The read-only cache optimization is discussed in section 5.5.

We will use the translation of a Mint-annotated 7-pt 3D stencil code implemented in C++ as a running example, illustrating the before-and-after, of the translation. Lastly, we will provide information about Spearmint's compiler options and installation process. An overview of Spearmint and its control-flow is illustrated in figure 5.1.

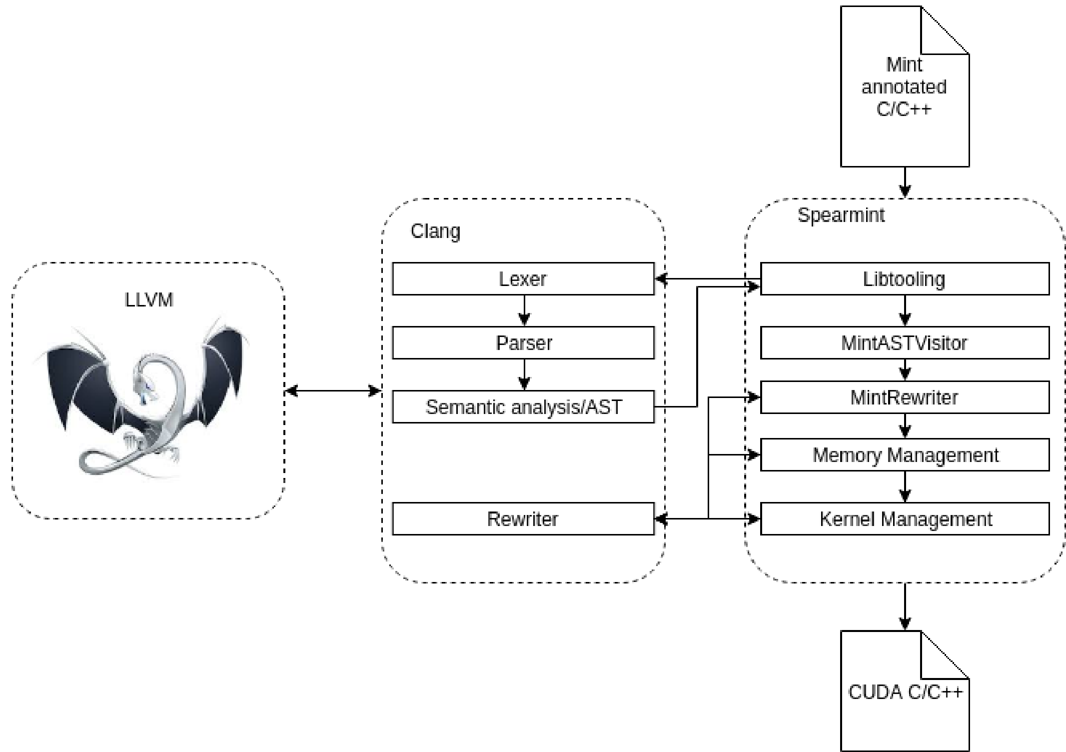

Figure 5.1: Overview of Spearmint's design and its control-flow.

5.1 Adding Mint Directives to Clang

In order to introduce Mint directives to Clang and allow creation of AST-nodes for the directives we had to make several extensions to Clang. We designed these extensions based on Clang's already-implemented OpenMP support. We added about 3700 lines of code to Clang, while implementing our modifications.

First, we modified Clang's basic library. We added Mint-annotations that Clang's pragma-handler/pre-processor later attaches to the start- and end-tokens of a Mint directive. It also includes defining Mint-specific enums, functions and statement nodes. Statement nodes (e.g. all nodes which are a sub-class of Clang's highest level statement.) are defined in a .def file. This file is later used by macros to auto-generate code for them. The RecursiveASTVisitor's Visit* function for AST-nodes discussed in subsection 4.2.1 is an example of one such case.

5.1.1 Modifying Clang's Parser

We have made modifications to Clang's parser library. Clang's parser library utilizes a pragma handler that handles all code-lines starting with "#pragma". By implementing and registering our own pragma handler, all pragmas starting with "#pragma mint" are directed to our handler. The pragma handlers have no way of creating AST-nodes directly, but they may annotate tokens[4, p. 111][1] in order to guide the parser. Our Mint pragma handler simply annotates the start and end (e.g. start and end of the line) of the pragma. When the parser later encounters those annotated tokens, it delegates those tokens to the Mint-related parsing functionality.

We have also implemented parsing functionality for the different Mint pragmas. The Mint parsing functionality is hand-written, as opposed to auto-generated parser code based on a Context-free Grammar (CFG). The Mint-parser gathers up information about source-location, identifiers, associated compound-statements and other relevant information defined through the Mint pragmas. The parser also performs syntactic checks, and if these checks pass, the gathered information is passed along to Clang's semantic library (libsema, see table 4.1).

5.1.2 Modifying Clang's Semantic Library

Clang's semantic library (libsema) performs semantic checks, and if the checks pass, the library calls appropriate create-methods in Clang's AST-library (libast) in order to create the AST-nodes. The need for explicit create-methods for AST-nodes is because Clang uses its own memory-allocator. This is a design decision to avoid memory-leaks and facilitates a unified way of allocating memory across the Clang-project. An AST-node's destructor is never called, all memory is simply free'd by the allocator. It is therefore important to allocate through the allocator, otherwise memory-leaks arise for the allocated memory that the allocator is not aware of.

[1] A token is an identifier, keyword, special characters, literals, etc.

5.1.3 Modifying Clang's AST Library

In order to represent the Mint directives in the AST, we have modified the AST-library. By adding our own AST-nodes to Clang's AST we can visit those nodes through the earlier discussed RecursiveASTVisitor (see subsection 4.2.1), or use Clang's ast-dump option to view the generated AST for a given source-file.

Our modifications to the AST-library include creating our own subclasses of the "Stmt" AST-node. In our AST-nodes we store source-locations, clauses specified for the directive, expressions and statements that specifies e.g. variable names and memory dimensions for the *copy*-directive. We also store pointers to associated-statements like a "ForStmt" in the case of a Mint *for* directive. All AST-nodes are required to implement some specific methods that allows querying of the nodes' type, accessors for the nodes' children in the AST, getters and setters for source-locations etc. These methods are required by several of the libraries that operate on the AST.

Clang's memory-allocation scheme influences how we store variable-length structures, for example lists of expressions, that are a part of the AST-node. We need to make special considerations in the AST-node's create-method when we are allocating memory from the allocator. We therefore allocate memory for both the AST-node's object and the variable-length structures in the same memory block, and store the variable-length structure directly after the AST-node's object. An example of such a variable-length structure is the list containing clauses for the Mint *for* directive.

We have also implemented our own representations of the Mint clauses, this is done in a similar manner to what is discussed above for the Mint directives.

In addition, we have also modified the AST-library to support pretty-printing, profiling and serialization of our AST-nodes. These additional modifications are not strictly necessary, but the functionality provided is nice to have.

5.2 Creating Spearmint with Clang's tooling library

After we implemented Clang support for Mint's directives we created our LLVM-based source-to-source translator, named Spearmint. Spearmint is a standalone program that utilize the Mint-modified Clang described in section 5.1.

In this and following sections we will describe in detail how our source-to-source translator is implemented. We will show step-by-step how a 7-pt 3D stencil code implemented in C++ with Mint annotations is translated to CUDA C++. Listing 5.1 shows an excerpt from the 7-pt stencil code. The excerpt contains the most interesting part from the stencil code and in this case the excerpt is the only part of the original source that is annotated with Mint directives.

```
1  #pragma mint copy(u_old, toDevice, (Nx+2), (Ny+2), (Nz+2))
2  #pragma mint copy(u_new, toDevice, (Nx+2), (Ny+2), (Nz+2))
3  #pragma mint parallel
4    {
5      for(int t = 0; t < num_iterations; t++) {
6  #pragma mint for nest(all) tile(16, 32, 64) chunksize(1, 1, 32)
       nowait
7        for (int k = 1; k < Nz+1; k++) {
8          for (int j = 1; j < Ny+1; j++) {
9            for (int i = 1; i < Nx+1; i++) {
10             u_new[k][j][i] = kC1 * u_old[k][j][i]
11               + kC0 * (u_old[k-1][j][i] + u_old[k+1][j][i]
12                       + u_old[k][j-1][i] + u_old[k][j+1][i]
13                       + u_old[k][j][i-1] + u_old[k][j][i+1]);
14           }
15         }
16       }
17       std::swap(u_new, u_old);
18     }
19 #pragma mint barrier
20   } // end of mint parallel
21 #pragma mint copy(u_old, fromDevice, (Nx+2), (Ny+2), (Nz+2))
```

Listing 5.1: Excerpt from a Mint annotated 7-pt 3D stencil code implemented in C++.

Spearmint utilizes several of Clang's libraries, some of which we

described in section 4.2. Additionally, Spearmint use Clang's lexing library (liblex), parsing library (libparse), semantic library (libsema), AST library and Clang's basic library (libbasic), several of which we have modified to accept Mint annotations. A brief description of what these libraries do can be seen in table 4.1. Spearmint also use some of LLVM's libraries directly, such as its system abstraction library (libsystem) and its basic support library (libsupport). Spearmint is implemented in C++ and is a standalone tool maintained in its own code base consisting of about 2200 lines of code. The relatively small code base is a direct result of utilizing LLVM and Clang in our project. Using LLVM as a library allowed us to avoid implementing common compiler functionality and focus on the core of our problem. We believe that Spearmint's small code base will make Spearmint easier to maintain and develop in the future.

Spearmint uses Clang's LibTooling library as its interface to Clang's functionality. In our case, using the LibTooling library involves an analyse-generate-modify approach. This means that we first traverse the generated AST and in the process gather information and analyse it. After the analysis stage we generate appropriate code based on our analysis. The generated code is internally represented as a std::string created using a string stream. The modify stage consists of inserting (or replacing) the generated code into the source input.

Spearmint's interface utilizes Clang's common options parser to parse command-line arguments. It then creates a *ClangTool* with the source-files and command-line arguments. Spearmint then creates a *MintFrontendAction* (a subclass of ASTFrontendAction). The MintFrontendAction implements a CreateASTConsumer() method which returns a MintAST-Consumer (a subclass of ASTConsumer). When the ClangTool is run, Clang lexes, parses, performs semantic checks and builds an AST for the given input source. Because of the Clang-modifications discussed in sub-section 5.1, any Mint directives in the input code appear in the generated AST. The MintASTConsumer then initiates the MintASTVisitor (a subclass of RecursiveASTVisitor) that visits all AST-nodes in a pre-order depth-first traversal. The MintASTVisitor implements special handling for the AST-nodes we are interested in, namely the Mint directives. The Mint-ASTVisitor simply populates a list with all Mint directives encountered in the traversal. The list of Mint directives is later iterated and depending on

the Mint directive, handled in a specialized way.

After the MintASTVisitor's information gathering on the AST is complete, control is given to the "MintRewriter" which performs general source-level editing like commenting out the directives discovered through the rewrite-library (librewrite). The MintRewriter then turns control over to the memory manager and kernel manager, respectively. We will go more detailed into the memory management and kernel management in the following subsections.

5.3 Memory Management

To access an array in a kernel run on a GPU, the array must first be copied to the GPU memory. CUDA C/C++ requires explicit programming in order to transport the arrays to the GPU, via the PCIe bus.

It is the Mint model's *copy* directive that informs the translator what arrays to transport between the host and the device as described in subsection 3.1.2. The directive's syntax is described in subsection 3.2.3.

Spearmint needs to reason about the memory-references inside a nested for-loop in order to correctly translate it. It may also need to do that in order to implement optimizations. Spearmint therefore requires the programmer to write array-accesses in a multidimensional form (e.g. array[z][y][x]), as opposed to a "flattened" (e.g. array[z * (height * width) + (y * width) + x]) access. As it is harder to recognize for example stencil patterns when the array accesses are "flattened". However, this should not be a huge imposition for the programmer, as it is generally easier to write multidimensional array-accesses while programming.

In order to allow the programmer to write multidimensional array accesses in the input source while at the same time utilizing hardware-aligned memory (pitched memory), the Mint model and Spearmint impose some restrictions on the way memory is allocated in the input source. Memory in the original source needs to be allocated in a contiguous block, otherwise Spearmint will not be able to correctly transfer it to the device. In order to be able to write multidimensional array-references in the input source, while at the same time having the memory allocated in a contiguous block, some additional memory is used. The additional memory is used for pointers into the contiguous

memory-block containing the actual data, allowing multidimensional array-references for what is in reality a one-dimensional block of memory. In listing 5.2, we provide a memory-allocation function that should be explicitly used in Mint-annotated code for memory that is to be transferred between the device and host. We also provide a function for deallocating

```
1  template<typename T> extern T*** alloc3D(const int x, const int y,
       const int z) {
2
3    T*** array3D = (T***)malloc(sizeof(T**) * z);
4    T** m_tempzy = (T**) malloc(sizeof(T*) * y * z);
5    T* m_tempzyx = (T*) malloc(sizeof(T) * x * y * z);
6
7    for (int i = 0; i < z; i++) {
8      array3D[i] = m_tempzy;
9      m_tempzy += y;
10     for (int j = 0; j < y; j++) {
11       array3D[i][j] = m_tempzyx;
12       m_tempzyx += x;
13     }
14   }
15   return array3D;
16 }
```

Listing 5.2: Recommended memory allocation code in C++ for code that is to be translated by Spearmint.

the memory allocated in this manner. The deallocating function can be seen in listing 5.3.

The discussed memory-scheme and that Spearmint allocates pitched memory on the device, impacts how we access device-memory. The reason for this is because the memory is laid out in a one-dimensional space on the device, and we therefore need to "flatten" the multi-dimensional memory accesses in our generated kernels. We will demonstrate how the array-accesses are "flattened" later in this section.

It is Spearmint's memory management library that handles the *copy* directives. When the memory manager is invoked it receives a list of all Mint directives found in the given translation unit. The memory manager traverses the list and visits each Mint copy directive. For each visited copy directive it creates a "CudaMemory_t"-struct that contains the generated names to be used for different variables associated with the

```
1 template<typename T> void free3D(T*** buffer) {
2   free(buffer[0][0]);
3   free(buffer[0]);
4   free(buffer);
5   buffer = nullptr;
6 }
```

Listing 5.3: Recommended memory deallocation code in C++ for memory allocated with the code provided in listing 5.2.

copy. The struct also contains the mapping between the host-pointers and the Spearmint-inserted device-pointers. This list is passed on to following translation stages.

After the initial analysis phase, Spearmint generates code that;

1. Defines and initiates a "cudaExtent" with the copy directives dimensions as arguments.

2. Defines a "cudaPitchedPtr".

3. Allocates memory through "cudaMalloc3D" and sends the "cudaPitchedPtr" defined in (2) and the "cudaExtent" from (1) as arguments.

4. Defines and initialize a "cudaMemcpy3DParms"-struct with the variables created in earlier steps and host-side variables as the structs fields.

5. Performs the memory-transfer through a call to "cudaMemcpy3D" with the "cudaMemcpy3DParms" defined in (4) as parameter.

When the copy-direction is from the device to the host, the steps are fewer. This is because Spearmint reuses the variables created when it generated the copy-code to the device. Spearmint simply redefines the fields of the "cudaMemcpy3DParms" struct created in (4). It swaps the source and destination fields of the struct and changes the "kind" field to reflect that the copy is in the device to host direction. Spearmint then generates code that calls "cudaMemcpy3D" with the redefined "cudaMemcpy3DParms" struct as parameter. If the Mint copy directive is found to be the last copy from device to host as seen in the collection of copy directives, Spearmint generates code that calls "cudaFree" on

all previously allocated device-memory, and with that, deallocating any remaining GPU memory.

```
1  /* Mint: Replaced Pragma: #pragma mint copy(u_old, toDevice, (Nx+2),
       (Ny+2), (Nz+2)) */
2    cudaExtent ext_dev_1_u_old = make_cudaExtent(((Nx + 2)) * sizeof(
        real ), (Ny + 2), (Nz + 2));
3  /* Mint: Malloc on the device */
4    cudaPitchedPtr dev_1_u_old;
5    cudaMalloc3D(&dev_1_u_old, ext_dev_1_u_old);
6  /* Mint: Copy to the device */
7    cudaMemcpy3DParms param_dev_1_u_old = {0};
8    param_dev_1_u_old.srcPtr = make_cudaPitchedPtr(((void *) u_old
        [0][0]), ((Nx + 2)) * sizeof(real), ((Nx + 2)), ((Ny + 2)));
9    param_dev_1_u_old.dstPtr = dev_1_u_old;
10   param_dev_1_u_old.extent = ext_dev_1_u_old;
11   param_dev_1_u_old.kind = cudaMemcpyHostToDevice;
12   cudaMemcpy3D(&param_dev_1_u_old);
```

Listing 5.4: Example memory-copy code generated by Spearmint.

The generated code is stored in a std::string. The string is passed to Clang's rewriter library (librewrite) that modifies the source-buffers to contain the contents of the generated string in the location specified. An example of copy-code generated for a Mint copy directive can be seen in listing 5.4. Line 1 shows the replaced Mint copy directive. The following lines show the steps discussed earlier in this subsection.

5.3.1 Error Handling

The code in listing 5.4 shows that Spearmint does not generate if-tests that check whether the return value from "CudaMemcpy3D" indicates success or failure. Didem Unat's Mint translator[61] generates such error-checks. In Spearmint we chose not to generate such error checking as it imposes extra library dependencies in the generated code. For example if the generated if-tests prints error messages through the *printf* function upon failure, the *stdio* library needs to be included. There are several ways to report failures, for example through logging libraries, streams, etc. Generating such code could create inconsistencies in how failures are reported in the overall program. However, the functionality is useful in itself. Future work may include implementing an option

to Spearmint which allows enabling of such error-checks, or possibly provides functionality that allows the user to define the way the error-checks should be performed.

5.4 Kernel Management

Spearmint's kernel management is tasked with handling kernel generation, kernel configuration and kernel execution. In this section we will describe the translation-steps performed by Spearmint in order to achieve this.

The kernel management receives the full list of Mint directives discovered in the given translation unit and the full list of "CudaMemory_t"-structs created in the memory manager. The list of "CudaMemory_t"-structs is needed to map between host-pointers and device-pointers inside the Mint *parallel* directive. Spearmint's kernel management breaks the translation job into several smaller tasks which we discuss in the following.

5.4.1 General Analysis

The first action Spearmint's kernel-management library performs is analysis of the *for* directives and their associated for-loops. The analysis step creates a CudaKernel_t-struct for each for directive encountered. This struct contains a pointer to the AST-node representing the Mint for directive and its clauses. It also contains the dimensional configurations for the kernel. The struct contains a mapping between the original array-accesses and the generated array-accesses to be used in the translated code. The map is populated during analysis, either with standard flattened array-references into global device memory or alternatively into read-only cache as a result of an optimization. Currently, the only implemented optimization that utilizes CUDA's on-chip memory is a read-only cache optimization. However, the translation map is intended to simplify implementation of register and shared memory optimizations as well. Other fields in the CudaKernel_t-struct also contains the generated name for the kernel, nesting-level of the kernel, a list of the parameters to the kernel and a list of all the for-loops in the kernel.

The list of the for-loops is a list of LoopBounds_t-structs, each containing the upper and lower bounds of the loop, the binary operator used in the original source code and a pointer to the AST-node representing the for-loop in the input source code.

The first step of the analysis is to access all Mint clauses specified for the for directive and use their values instead of the default values. Spearmint then generates a name for the new kernel. The naming convention used is an integer counter starting at one that grows upwards for each kernel created. This integer is prefixed with *mint_* and given *_1527* as a suffix. As an example, the first kernel created for any translation unit will be named *mint_1_1527*. Spearmint does not currently check if the generated name already exists in the translation unit, so names following this convention should be avoided in the source-input.

The analysis proceeds with finding all externally declared variables. Externally declared variables include global variables, which in normal C/C++ do not need to be passed to a function because the function would have direct access to them. However, in the case of CUDA, where the kernel is executed on a GPU, all accessed data must be sent to the GPU over the PCIe bus. Data must be passed over the PCIe bus either implicitly as parameters to a kernel (scalar types, including pointers), or through explicit-copying in the case of arrays[14, section 3.2.2]. In the case of explicit copying, a pointer referencing the explicitly-copied memory must be sent as a parameter to the kernel invocation. The definition of an externally declared variable is all the variable references that is not declared inside an accelerated region. More formally the set of externally declared variables can be defined as: $P = R - V$. Where R is the set of variable *references* inside the for directive and V is the set of variable *declarations* inside the for directive. The resulting set P contain the variables that need to be passed to the kernel as parameters. Spearmint checks that all externally declared arrays used inside a *for* directive are copied in with the *copy* directive and if they are not, Spearmint prints an error message and exits the translation.

The analysis continues to find the upper- and lower-bounds of the nested for-loops in the kernel and populates the list discussed previously in this subsection. The analysis-phase checks the validity of the kernel configurations. The checks entail making sure the number of threads

per block does not exceed 1024 threads (current maximum), and check that there are no more than 64 threads per block in the Z-dimension (current maximum). If these checks fail, Spearmint prints appropriate error messages and exits the translation.

Spearmint then prints the resulting name, nesting-level, configuration, parameter-list etc. of the kernel to be generated to the command line. The output generated by Spearmint while translating the 7-pt stencil code in listing 5.1 can be seen in listing 5.5. When the analysis is

```
1 INFO:Spearmint: Using following configuration for Mint 'for' @ Line
     111:9
2 Nesting level: 3, Tile Sizes: (16, 32, 64), Thread geometries: (16,
     32, 2), Chunk sizes: (1, 1, 32)
3 Nowait: true
4 Number of threads per block: 1024
5
6 INFO:Spearmint: Generating a kernel with name: mint_1_1527
7 INFO:Spearmint: Arguments passed to the kernel: { Nz, Ny, Nx, u_new,
      kC1, u_old, kC0}
```

Listing 5.5: Example output from the analysis phase.

completed the information gathered is inserted in a list containing all the "CudaKernel_t"-structs and this list is used by the following translation steps.

5.4.2 Analysing the memory accesses

Following the general analysis described in subsection 5.4.1 there comes analysis of the memory accesses. By analysing the access pattern and the number of reads and writes to each array we can optimize the kernel code by choosing the appropriate device-memory for the different arrays. Shared memory in CUDA is beneficial for memory addresses that are being accessed by several threads in a thread-block. Registers are beneficial for memory addresses that are accessed a high number of times by the same thread on GPUs that do not cache global memory accesses. Read-only cache on newer CUDA GPUs (compute capability 3.5 and higher) is beneficial for memory addresses that are read by several threads in a thread-block but never written to. As the read-only cache never is written to, inter thread-block synchronization is unnecessary, and

seeing as there always will be a performance cost with synchronization, utilizing read-only cache may often be preferable compared to shared memory. Additionally, utilizing the read-only cache path takes pressure off of the L1 cache/shared memory data path and the read-only cache has higher bandwidth than that of the L1 cache/shared memory[42].

The memory-access analysis start by traversing the for-loops following a for directive and all of its sub-trees. During the traversal all memory accesses to arrays are collected into two map structures. After the traversal, one map contains all memory-reads and the second map contains all memory-writes. The memory accesses are stored with the identifier of the memory as the entries key and a list of all the array subscript expressions for the identifier as the entries value. An example of the contents of the two maps can be seen below.

- Read
  - $u_old \rightarrow \{u_old[i][j][k], u_old[i][j][k+1], u_old[i][j][k-1], ...\}$
  - $rhs \rightarrow \{rhs[i][j][k]\}$
- Write
  - $u_new \rightarrow \{u_new[i][j][k]\}$

Note that we keep *all* memory references in the maps, even identical ones. The knowledge of how many times a distinct memory address is accessed may be used for register optimization for example, lowering pressure on the caches. These two maps are currently used by our read-only cache optimization, and are intended to be used by other optimizations as well.

Since the arrays are one-dimensional on the device, the array references must be *flattened* to reflect that in kernel code. An array reference such as *u_old[k][j][i]* is flattened to *u_old[u_old_index]*. Both of these different array accesses access the same element in host-code and device-code, respectively. Here *u_old_index* represents the stencil's central point and is equivalent to using the multidimensional u_old[k][j][i]. How the stencil's central point is computed is described in subsection 5.4.7.

In cases where a multi-dimensional array-access contains an offset in one or more dimensions, Spearmint use *width* and *slice* values to offset the central point in the y-axis and z-axis, respectively. In cases where the offset

is in the x-axis, the offset simply remains the same, as the elements in the x-axis lies contiguously in memory.

An array's central point, its *width* and its *slice* values used to compute the flattened index are computed at the start of a kernel and placed in registers. We will discuss how these values are computed and what they represent in subsection 5.4.7. The flattened array references are placed in the translation table discussed in subsection 5.4.1, to be used in later translation stages.

5.4.3 Replacing Externally Declared Arrays

After the analysis the kernel manager proceeds with replacing all externally declared arrays inside a *parallel* directive with their cudaPitchedPtr counterpart. An example of this, is a reference to an externally declared array named *u_old*. *u_old* will be swapped out with *dev_1_u_old.ptr* in order to perform the action from the original source-input on the device-pointer instead. If the reference to the externally declared array is used as an initializer to a variable declaration, Spearmint changes the variable declarations type in order to reflect the change. The *ptr* field of the cudaPitchedPtr struct is of type *void**, while externally declared arrays can be of several different types like *float*** and *double****. The call to std::swap, on line 17 in listing 5.1 will be changed to the code seen in listing 5.6.

```
1 std::swap(dev_2_u_new.ptr, dev_1_u_old.ptr);
```

Listing 5.6: An example replacement of an externally declared array. The code shown is a result of translating line 17 in listing 5.1.

5.4.4 Inserting Forward Declarations of Kernels

The kernel manager then inserts forward declarations of the kernels to be generated into the source-buffers. The MintVisitor we discussed in 5.2 passes a reference to the last function declaration encountered before the first Mint directive is discovered, to the kernel manager. The kernel manager inserts the forward declarations directly before that function declaration. This ensures that there will be no Mint directives, and following that, no calls to a Mint kernel before the forward declarations.

An example of a forward declaration generated by Spearmint may be seen in listing 5.7. A CUDA kernel can have no return value, hence the use of *void*.

```
1 __global__ void mint_1_1527(int Nz, int Ny, int Nx, cudaPitchedPtr
    dev_2_u_new, real kC1, cudaPitchedPtr dev_1_u_old, real kC0);
```

Listing 5.7: An example of a forward-declaration of a kernel generated by Spearmint.

Note the function type qualifier *__global__* of the kernel declaration in listing 5.7. The type qualifier signals that the function is a CUDA kernel that is executed on the device. It also signals that the kernel is callable from the host and that its execution is asynchronous, meaning the kernel-call returns before the device has completed the kernel execution[14, Appendix B.1.2.].

5.4.5 Inserting Kernel Configuration Code

The kernel manager then generates and inserts kernel configuration code. Kernel configuration code computes the number of thread-blocks to be created in each dimension and number of threads in each dimension of a thread-block. These computations are stored in a *dim3*-struct which is later passed to the kernel execution. The Spearmint-generated kernel configuration code for the 7-pt 3D stencil code in listing 5.1 can be seen in listing 5.8.

```
1   /* Kernel execution configuration for kernel: mint_1_1527 */
2   int num3blockDim_1_1527 = (Nz + 1 - 1) % 64 == 0 ? (Nz + 1 - 1) /
       64 : (Nz + 1 - 1) / 64 + 1;
3   int num2blockDim_1_1527 = (Ny + 1 - 1) % 32 == 0 ? (Ny + 1 - 1) /
       32 : (Ny + 1 - 1) / 32 + 1;
4   int num1blockDim_1_1527 = (Nx + 1 - 1) % 16 == 0 ? (Nx + 1 - 1) /
       16 : (Nx + 1 - 1) / 16 + 1;
5   dim3 gridDim_1_1527(num1blockDim_1_1527, num2blockDim_1_1527,
       num3blockDim_1_1527);
6   dim3 blockDim_1_1527(16, 32, 2);
```

Listing 5.8: An example of kernel configuration code generated by Spearmint for the 7-pt 3D stencil code in listing 5.1

In the case of the generated code in listing 5.8, the kernel configuration generated is for a *for* directive with nesting level 3. Alternatively, the *nest* clause was used with the keyword *all* and Spearmint discovered three nested for-loops eligible for parallelism. Of the for-loops to be parallelized, Spearmint always choose the innermost one as the x-dimension. If there are two or more loops to be parallelized the for-loop directly outside of the innermost one becomes the y-dimension. If the nesting-level is three, the outermost for-loop becomes the z-dimension.

As a small optimization for the cases where a Mint for directive is inside a loop, the kernel configuration code is inserted directly before the entrance to the parallel-block. Moving the kernel configuration code outside the parallel-block ensures that the computation is performed only once, instead of once for every iteration in the surrounding loop.

5.4.6 Inserting Kernel Execution Code

The kernel manager then generates and inserts kernel execution code. CUDA kernel execution is a regular C/C++ function call, with some extensions to the syntax. Listing 5.9 shows the Spearmint generated kernel-call for the 7-pt 3D stencil code in listing 5.1. Between the <<< and the >>> on line 2, the kernel configuration computed in listing 5.8 is inserted. This tells the CUDA runtime how many thread-blocks in each dimension and how many threads per block in each dimension we want it to use in our computation, respectively.

Here we may also specify how much dynamically shared memory each block in the kernel should be given and which CUDA-stream to execute the kernel in[14, Appendix C.3.1.1]. Currently, Spearmint does not use these options. However, one can use CUDA-streams to overlap memory movement on the memory bus and computation. Additionally, CUDA-streams can be used for parallel memory movement to the device and from the device simultaneously, as the PCIe bus support full-duplex[48]. This is something we leave as future work.

Note the use of the *nowait* clause on Line 1 in listing 5.9. If the nowait clause is omitted, Spearmint would insert a function-call to *cudaDeviceSynchronize* in order to synchronize the host and the device after the kernel-call. Spearmint does not automatically insert synchronization

```
1 /* Mint: Replaced Pragma: #pragma mint for nest(all) tile(16, 32,
     64) chunksize(1, 1, 32) nowait */
2   mint_1_1527<<<gridDim_1_1527, blockDim_1_1527>>>(Nz, Ny, Nx,
      dev_2_u_new, kC1, dev_1_u_old, kC0);
```

Listing 5.9: An example kernel execution call generated by Spearmint for the 7-pt 3D stencil code in listing 5.1

calls where needed, when the nowait clause is used. If the programmer utilizes the nowait clause he/she must insert the Mint *barrier* directive where explicit synchronization is needed.

Prefer L1 Cache Configuration

If Spearmint is run with the -preferL1 flag, Spearmint inserts a function-call to *cudaFuncSetCacheConfig()* with the name of the kernel and the *cudaFuncCachePreferL1* enum as parameters before the kernel execution. This makes the compiler configure the L1 cache so that 48KB is used for L1 cache and 16KB is used for shared memory.

The default configuration favours 48KB of shared memory and 16KB of L1 cache, so we do not provide a optimization flag that sets this cache configuration as it is implicit.

5.4.7 Kernel Creation

The last and most involved step in the translation is to generate the kernels themselves. The code that generates the forward declaration for the kernel is reused while creating the kernel-signature. Spearmint starts with unpacking the *cudaPitchedPtr*-struct and computes the *width* and *slice* for the respective memories. Here width means the width of the actual allocated memory in the x-axis, and not the number of elements in the x-axis. While using pitched memory the CUDA runtime may allocate more memory in the x-axis than is needed. This is because of hardware memory-alignment reasons. Aligned memory makes memory accesses faster because the accesses more often becomes coalesced[13, section 9.2.1]. A slice is the width multiplied by the size of the array's y-axis. It is not the amount of elements in the $x * y$ square, but it is the amount of actually allocated memory. The width and slice are needed for

pointer arithmetic and are used while flattening the array accesses from multi-dimensional to one dimensional, as discussed in subsection 5.4.2. Listing 5.10 lines 1 through 7 show the generated kernel definition, unpacking of the cudaPitchedPtr-struct and calculation of the width and slice for the pitched memory.

As a small optimization Spearmint searches the collection of arrays that are copied to the device and looks for arrays with the same dimensionality and type. When two or more arrays with the same dimensions and type are found, Spearmint groups them together and use the same computed width and slice for the arrays that match.

We have tried to make the generated code similar to the original source and have reused the original names from the source-input. However, we have augmented them by prefixing *_width_* and *_slice_* to the original name of the array for the array's width and slice, respectively. For arrays that match with other arrays in type and dimensionality, a number signifying the group of matching arrays are used instead of the array's name.

Spearmint then generates code that calculates the indexes for the thread in each dimension. The generated index computation can be seen in line 9-11 in listing 5.10. The computations differ for the dimensions where the thread only computes one element (lines 10 and 11), compared to the dimensions where the thread computes several elements (line 9). In cases where a thread computes several elements in a dimension, the computation needs to take that into account in order to correctly offset the index. Spearmint supports chunking in all three dimensions and in several dimensions at once. However, chunking in several dimensions often hurts performance as it decreases parallelism, so chunking should be used with care.

When several elements are computed by a thread, special considerations have to be made when computing the generated for-loops upper-bound. The generated code that computes a for-loop's upper-bound may be seen on line 13 in listing 5.10. The generated code sets the upper-bound to be either the number of elements to be computed in the given dimension (chunksize), or the original upper-bound of the translated for-loop to avoid computing too many elements. A situation where you could risk computing too many elements is when the loop-range for a for-loop in the

```
1  __global__ void mint_1_1527(int Nz, int Ny, int Nx, cudaPitchedPtr
       dev_2_u_new, real kC1, cudaPitchedPtr dev_1_u_old, real kC0) {
2
3    real *u_new = (real *) dev_2_u_new.ptr;
4    real *u_old = (real *) dev_1_u_old.ptr;
5
6    int _width_0 = dev_2_u_new.pitch / sizeof( real);
7    int _slice_0 = dev_2_u_new.ysize * _width_0;
8
9    int k_idx = threadIdx.z * 32 + (blockDim.z * 32 * blockIdx.z) + 1;
10   int j_idx = threadIdx.y + (blockDim.y * blockIdx.y) + 1;
11   int i_idx = threadIdx.x + (blockDim.x * blockIdx.x) + 1;
12
13   int k_upperBound = k_idx + 32 < Nz + 1 ? k_idx + 32 : Nz + 1;
14
15   int j = j_idx;
16   if(j >= 1 && j < Ny + 1) {
17     int i = i_idx;
18     if(i >= 1 && i < Nx + 1) {
19       for(int k = k_idx; k < k_upperBound; k += 1) {
20         int _index_0 = i + j * _width_0 + k * _slice_0;
21         u_new[_index_0] = kC1 * u_old[_index_0] + kC0 * (u_old[
               _index_0 - _slice_0] + u_old[_index_0 + _slice_0] + u_old[
               _index_0 - _width_0] + u_old[_index_0 + _width_0] + u_old[
               _index_0 - 1] + u_old[_index_0 + 1]);
22       }
23     }
24   }
25 }
```

Listing 5.10: A CUDA kernel generated by Spearmint for the 7-pt 3D stencil code in listing 5.1.

original code is not divisible by the size of the thread-block in the same dimension. In such a case an extra thread-block is added to compute the remaining elements. At least one or more of the threads in that thread-block will be superfluous and must be kept from performing computation.

The kernel management then generates either if-guards that makes sure a thread's index is within the ranges of the original for-loop or a for-loop computing the specified number of elements in a given dimension. As a small optimization, the if-guards generated, if any, are placed outside (before) the generated for-loops to avoid doing the same check several times. The generated if-guards and for-loops for a kernel can be seen on

line 15-19 in listing 5.10.

In cases where the number of for-loops in the loop-nest exceeds the nesting-level, Spearmint inserts the for-loops in the original code as-is. These loops will be nested inside the generated if-guards or for-loops for the parallelized loops.

On line 20 in listing 5.10 Spearmint has generated code that computes the array's *central-point* index. Spearmint use the central-point index to flatten multidimensional array-accesses, as discussed in subsection 5.4.2. The computation of the central-point's index is based on the earlier discussed width and slice and is only computed once per array or once per group of arrays that have matching type and dimensions.

Spearmint then searches the AST for all occurrences of array-references and looks up their translation in the CudaKernel_t-struct's translation table discussed in subsection 5.4.1. The translated array-reference found in the translation table is used in the original array-reference's place. Line 21 in listing 5.10 shows the resulting translation done by Spearmint of line 10-13 in listing 5.1.

5.5 Read-Only Cache Optimization

There are several optimizations we could have implemented in our source-to-source translator. The Mint translator described in section 3.3 implement shared memory and register optimization. We wanted to implement optimizations that are specific for NVIDIA's Kepler architecture and newer.

We have implemented an optimization that utilize read-only cache in our source-to-source translator. CUDA's read-only cache is a constant cache that was introduced with the Kepler architecture and is implemented on devices of compute capability 3.5 and higher. The cache is on-chip and is accessible by all threads in a thread-block. The data cached in the read-only cache is valid for the lifetime of a kernel.

When this optimization is switched on with the -readOnlyCache flag sent to Spearmint, this optimization step is performed directly after the analysis of the memory accesses discussed in subsection 5.4.2. Our read-only cache optimization is quite simple. The optimization search through the two maps discussed in 5.4.2, that contain the memory accesses for a

given kernel. It selects all memory references to an array that only appear in the map containing memory *reads* as candidates for the read-only cache. More formally it can be described as $C = R - W$. Where R is the set containing arrays that are *read* from and W is the set of arrays that are *written* to. The resulting set C contains the arrays eligible for the read-only cache.

All arrays in the resulting set C can be cached in the read-only cache without it leading to errors, as they are never written to. However, we are only interested in caching memory reads that are shared between threads in a thread-block, so we only select the arrays that are referenced more than once.

The optimization then wrap all memory references to selected arrays with the *__ldg()* intrinsic, ensuring that the memory accesses are loaded through the read-only data cache. The memory references wrapped with the __ldg() intrinsic are then placed in the translation table for that kernel and used in the kernel creation step discussed in subsection 5.4.7. Listing 5.11 show the code inserted in the place of line 23 for the 7-pt 3D stencil code shown in listing 5.10.

```
1 u_new[u_new_index] = kC1 * __ldg(&u_old[u_old_index]) + kC0 *
2   (__ldg(&u_old[u_old_index - u_old_slice]) +
3   __ldg(&u_old[u_old_index + u_old_slice]) +
4   __ldg(&u_old[u_old_index - u_old_width]) +
5   __ldg(&u_old[u_old_index + u_old_width]) +
6   __ldg(&u_old[u_old_index - 1]) +
7   __ldg(&u_old[u_old_index + 1]));
```

Listing 5.11: The resulting computation for line 23 in the 7-pt 3D stencil code in listing 5.10 with read-only cache optimization.

The optimization does not add any branching or explicit synchronization calls in the generated code, in the same way Mint's shared memory optimization does, as discussed in section 3.3.

Our optimization greedily puts all arrays eligible for read-only cache with more than one access to that array in the read-only cache. This is perhaps naive because there are several considerations to take into account while using the read-only data cache. The read-only cache is only 48 KB per SM for the Kepler architecture, making it a scarce resource. The greediness of the current implementation may lead to overuse of the read-

only cache and lead to a high number of cache misses, which in turn causes global memory accesses.

Possibly a better way of selecting the memory to put in the read-only data cache would be to prioritize which memory to put in the cache, and only put a limited amount in it. Highest priority should be given to the memory that would result in decreasing global memory accesses the most while at the same time keeping cache misses to a minimum by avoiding overuse. Such prioritized selection would ideally also take the size of a thread-block and the estimated number of thread-blocks running parallel on a SM into consideration, when determining how much memory to place in the cache. In order to achieve this, Spearmint would need hardware-specific knowledge about the GPU to generate CUDA code for.

Coupling this optimization with shared memory optimization might result in performance benefits as it could provide a total of 96KB of on-chip memory where some of the stencil data can be placed in the read-only data cache and some in shared memory.

However, due to time-restrictions and the already extensive work put into this project, we reserve further improvements to the optimization, as well as implementing other optimizations as future work. We will describe future work in section 8.1.

5.6 Spearmint's Compiler Options

Spearmint uses Clang's common options parser. The common options parser provides several common options by default. The default options includes specifying a list of input-files, printing help-text and specifying a compile database, which are provided out-of-the-box[34]. It also provides a *version* flag that we have expanded to include information about the Spearmint version.

In addition to the default options, we have added our own custom options to Spearmint, these can be seen in table 5.1

-o=<filename> Allows the user to specify the name of the output-file the generated CUDA C/C++ is written to. Default output file-name is the name of the input file with a .cu suffix.

-mintDebug Turns on debug printing. Intended to be used by Spearmint developers.

-silent Instructs the translator not to print information about the translation.

-preferL1 Instructs the translator to generate code that prefers a larger L1 cache.

-shared Instructs the translator to generate CUDA code that utilizes shared memory. Note that shared memory is currently unimplemented.

-readOnlyCache Instructs the translator to generate CUDA code that utilizes read-only data cache through CUDA's __ldg intrinsic. Note that this optimization only works on compute capability 3.5 and higher and the generated code must be compiled with e.g. the *-arch=sm_35* flag sent to nvcc for devices of compute capability 3.5.

Table 5.1: Spearmint's translation options.

5.7 Installing Spearmint

One of the major goals behind Spearmint was to make the installation as simple as possible. Part of the reasoning behind that goal was based on the fact that our target-user should not have to be an expert programmer. Another reason is that the Mint translator received negative feedback on its installation process and we wanted to rectify that problem. We also believe that when the installation process is as simple as possible this lowers the threshold for new users to start using our translator, resulting in wider adoption.

Our solution to this goal is in part achieved by using the LLVM compiler infrastructure. Additionally we provide an install-script that handles the entire installation from start to finish, with no user intervention. The user needs only download the script and run it, in order to install Spearmint. To update Spearmint to the latest version, simply run the install-script again and it will fetch and build the latest changes, if any. The script is currently tested on Linux, Mac and Windows 10 using Cygwin[17]. Initial feedback is positive.

The script is fairly simple, spanning over 36 lines of BASH code (with comments). The script starts with checking out the source code for LLVM,

the Mint-modified Clang and Spearmint, which are separated into three different projects/code bases. LLVM's source code is checked out from the LLVM project's Subversion repository residing on LLVM's home pages (http://llvm.org/svn). The source code for the Mint-modified Clang and for Spearmint are fetched from Git repositories residing in my Bitbucket account, where they are freely available as open source:

Spearmint https://bitbucket.org/niklasja/spearmint.git

Mint-modified Clang https://bitbucket.org/niklasja/clang_mod_lite.git

The install script (*install.sh*) is located in the root directory of the Spearmint repository and can be copied or downloaded from there.

The LLVM project has a nested project structure where Clang is placed inside of LLVM and Spearmint is placed inside Clang, this placement is performed automatically by the build script. The build script then generates *Unix Makefiles* build files by running CMake in LLVM's root directory. Finally the script builds all of the source by running *make*. After the install script is run, there will be two new directories in the root directory. The *llvm* directory that contains all the source code for all three projects and the *llvm_build* directory contains all of the binaries built in a subdirectory named *bin*. In order to verify the installation the command "spearmint –version" may be run and if the installation was successful the command will print version information.

For the more advanced user the script should be easily configurable if the user wants to use another build system than Unix Makefiles or wants to use different CMake configurations, for example. In such a case, knowledge about BASH, CMake and the selected build system are advantageous.

The system requirements for installing Spearmint are the same as they are for LLVM and Clang, so Spearmint may be used on all systems where LLVM and Clang can be used[22].

An installation of Spearmint consists of LLVM, a Mint-modified version of Clang, and Spearmint itself as a standalone Clang tool. Making modifications to Clang itself is sub-optimal as we will need to follow Clangs development tightly and merge our modifications into newer versions in order to benefit from Clang's future improvements.

Well into this project we discovered Clang Plugins[12]. Clang Plugins may be used to define pragmas by creating and registering a PragmaHandler. It may be worth investigating if it is feasible to move the functionality we added to Clang into Spearmint by using Clang Plugins. In doing that we could get a looser coupling to Clang and simplify the maintenance efforts related to Clang updates. As a side-effect, moving the Clang modifications to Spearmint would eliminate the need for having two versions of Clang installed on the system of a user who already has Clang installed. In cases where the user already has Clang installed, only Spearmint would then need to be installed, given that the version of the Clang installation is compatible with Spearmint. However, because of time-constraints, this remains as future work.

5.8 Summary

In this chapter we have presented Spearmint, our source-to-source translator based on the LLVM compiler infrastructure. The LLVM compiler infrastructure, Clang and the libraries and tools Spearmint are based on are described in chapter 4. Spearmint translates Mint-annotated C and C++ code into parallelized CUDA C/C++. As we have discussed the different parts of Spearmint in this chapter, we have provided code illustrating the before-and-after of the translation of a Mint-annotated 7-pt 3D stencil code implemented in C++.

In subsection 5.1 we discussed the modifications we made to Clang in order to include Mint's directives into the generated AST.

In sections 5.2-5.4, we described how Spearmint is created with Clang's libtooling library, Spearmint's memory management and Spearmint's kernel management, respectively.

In section 5.5 we discussed our read-only cache optimization. This optimization utilize newer CUDA GPU's read-only cache for data that remains constant throughout the lifetime of a kernel. However, our implementation have potential for improvements. Additionally, further work should include implementation of shared memory and register optimizations as well as optimizations that utilize intrinsics in the newer CUDA GPUs. There have for example been reported good performance results by utilizing CUDA's newer *shuffle* intrinsic while optimizing real-

world flood simulations[24].

We believe that our source-to-source translator is a good starting point for further research into automatically generated optimizations for CUDA GPUs. The simplicity and understandable code base of Spearmint coupled with the fact that Spearmint only consist of about 2200 lines of code (Mint-modified Clang excluded), should make Spearmint easy to work with.

In section 5.7 we outlined the installation process behind Spearmint. Spearmint was made with the intention that it should be easy to install and use. In order to achieve this goal we have based Spearmint on LLVM and Clang and we provide an installation script that performs the installation without any required user intervention.

It is worth noting that in the moment of writing, Spearmint should be considered experimental and that there probably exist several bugs and errors. One should therefore be extra careful and compare the results of the GPU computations with the results from running the non-Mint equivalent code on the CPU (where possible) to ensure correctness.

Chapter 6

Evaluating the Implementation

In this chapter we evaluate the performance of the Spearmint generated CUDA code. We start by describing our testing environment in section 6.1. In section 6.2 we describe the three common stencil codes we have chosen to perform our tests with.

In section 6.3 and 6.5 we compare the performance achieved by using the Mint translator and Spearmint, without optimizations and with optimizations, respectively. Generated code by the Mint translator with optimizations has been shown to be comparable to hand-optimized codes on Nvidia's Fermi architecture[61, chapter 6]. However, Mint's optimizations does not necessarily perform as well on the newer Kepler architecture and we would like to investigate how Spearmint's read-only cache performs on Kepler, in comparison.

In section 6.4 we provide a brief discussion about the performance differences incurred by the PreferL1 configuration option discussed in 5.4.6. In section 6.6, we provide a brief discussion about the code quality and usability delivered by the translators.

6.1 The Testing Environment

We ran our tests on Simula Research Laboratory's Lizhi machine located at Fornebu. Lizhi runs Ubuntu 12.04 LTS on two Intel Xeon E5-2650 processors and has 32GB of DDR3-1600 memory. Lizhi is connected to two Nvidia Tesla K20m GPUs through PCIe 2.0. In our tests we only utilize one CPU and one of the GPUs as the Mint model and our translator do

not support multi-GPU computation.

The Tesla K20m card is based on Nvidia's Kepler architecture and the card's CUDA compute capability is 3.5. On the card there is a GK110 chip consisting of roughly 7.1 billion transistors. On the K20m card there is 5GB of GDDR5 memory. The GDDR5 memory is connected through a 320-bit interface and the theoretical peak memory bandwidth is 208GB/s.

The GK110 chip contains 13 SMs (or SMXs in the case of Kepler), each with 192 cores giving 2496 cores in total with a clock cycle at 708MHz. Each SM has 65536 32-bit registers and a 64KB L1 cache where some of the memory in the cache may be used as software-configured shared memory. The L1 cache may be configured so that 16KB, 32KB or 48KB are used for shared memory and the remainder as a regular hardware managed L1 cache. Additionally, each SM has 48KB of read-only cache that can be either managed automatically by the compiler or explicitly by the programmer. The K20m's theoretical peak double-precision performance is 1.43 TFLOPS and provides a theoretical peak single-precision performance of 4.29 TFLOPS.

Lizhi's Tesla K20m GPUs have Error-Correcting Code (ECC) turned on, reducing the effective memory bandwidth with approximately 20%[13, section 8.2.1].

6.2 The Test Codes

To perform our tests and compare Spearmint with the Mint translator we used three common stencil codes that are applied to a regular Cartesian grid. All of the stencils are three-dimensional and all adhere to the limitations specified in section 2.2. The stencil codes selected are well researched and Didem Unat performed her tests of the Mint translator with these stencils in her PhD thesis[61]. Table 6.1 shows a summary of the stencils we used for testing. u denotes the regular Cartesian grid. The superscript n denotes the time step/iteration of the computation and the subscript (i, j, k) denotes the spatial index. Constant coefficients are denoted with c (with a subscript if more than one) and k and b denote two arrays containing variable coefficients or a given right-hand side vector.

The first 7-point stencil can be used to solve e.g. a heat equation that simulates how heat disperses in a three-dimensional object. This stencil

Stencil	Mathematical description	In, Out arrays	Read, Write per point	Operations per point
3D Heat 7-pt	$u^{n+1}_{i,j,k} = c_1 u^n_{i,j,k} + c_2 \left(u^n_{i,j,k\pm1} + u^n_{i,j\pm1,k} + u^n_{i\pm1,j,k} \right)$	1,1	7,1	2(*), 6(+)
3D Heat 7-pt variable coefficient	$u^{n+1}_{i,j,k} = u^n_{i,j,k} + b_{i,j,k} + c \left[k_{i+\frac{1}{2},j,k} \left(u^n_{i+1,j,k} - u^n_{i,j,k} \right) - k_{i-\frac{1}{2},j,k} \left(u^n_{i,j,k} - u^n_{i-1,j,k} \right) + k_{i,j+\frac{1}{2},k} \left(u^n_{i,j+1,k} - u^n_{i,j,k} \right) - k_{i,j-\frac{1}{2},k} \left(u^n_{i,j,k} - u^n_{i,j-1,k} \right) + k_{i,j,k+\frac{1}{2}} \left(u^n_{i,j,k+1} - u^n_{i,j,k} \right) - k_{i,j,k-\frac{1}{2}} \left(u^n_{i,j,k} - u^n_{i,j,k-1} \right) \right]$	3,1	15,1	7(*), 19(±)
3D Poisson 19-pt	$u^{n+1}_{i,j,k} = c_0 \left[b_{i,j,k} + c_1 \left(u^n_{i\pm1,j,k} + u^n_{i,j\pm1,k} + u^n_{i,j,k\pm1} \right) + u^n_{i\pm1,j\pm1,k} + u^n_{i\pm1,j,k\pm1} + u^n_{i,j\pm1,k\pm1} \right]$	2,1	19,1	2(*), 18(+)

Table 6.1: A summary of the stencil codes used for testing. For brevity we use $\pm$ to symbolize $u^n_{i,j,k\pm1} = u^n_{i,j,k-1} + u^n_{i,j,k+1}$ and $\frac{1}{2}$ to symbolize $k^n_{i,j,k+\frac{1}{2}} = k^n_{i,j,k} + k^n_{i,j,k+1}$.

uses two arrays, one for writing new calculated values to, and one for reading old values from. These two arrays are swapped with each other between two iterations where the values calculated in iteration n is used as the previous values in iteration $n+1$.

The second 7-point stencil can be related to a heat equation that has a source term and uses variable coefficients. This stencil code uses two additional arrays compared to the first stencil introduced. As seen in table 6.1, this stencil has a higher read-per-write ratio, as well as performing a higher number of arithmetic operations per point, compared to the 7-point heat stencil.

The third and final code is a 19-point stencil for solving the 3D Poisson equation. The Poisson stencil reads from two arrays and writes to one, using a total of three arrays. The 19-point Poisson stencil is the stencil with the highest read-to-write ratio with 19 unique reads per write, and performs fewer arithmetic operations than the 7-point variable coefficient stencil.

As a small optimization all codes utilize the *nowait* clause with Mint's *for* directive and explicit synchronization is inserted through the use of the *barrier* directive before the exit of the *parallel* directive. An example of how we have used the nowait clause in the test codes can be seen in listing 5.1, line 6. Line 19 shows the inserted barrier directive that ensures host-device synchronization before the memory is copied back to the host.

Using the nowait clause in this way will always be beneficial for our test-codes independent of different configurations and optimizations as it eliminates unnecessary host-device synchronization explicitly inserted by the translators.

During our tests all stencils are applied to a regular Cartesian grid with a size of 512^3, and 200 iterations/time steps. In order to compare the results of the Spearmint generated code with the code generated by the Mint translator, all test codes are written in C, as it is the only language Mint supports. One may argue that there are little performance differences between C++ and C in these minimalistic test-cases, however, there may be compiler differences giving one language an advantage over the other. All codes are compiled with Nvidia's nvcc compiler with the -O3 and -arch=sm_35 flags. Mint version 0.9.7.20 released with the Rose compiler infrastructure was always used. All codes generated by Spearmint are generated by version 1.0.7.

All kernels were measured five times on Lizhi, and we report the best performance observed out of those. The measurements exclude the PCIe transfer cost, meaning we only measure the performance of the kernels. Including the PCIe transfer cost would lower the reported GFLOPS performance, depending on the number of iterations/time-steps. By increasing the number of iterations, the impact of the PCIe transfer cost has on performance will decrease since the transfer cost is constant for our test codes.

6.2.1 Tile and Chunksize Configurations

In CUDA, different kernel configurations such as the number of threads per thread-block and the number of points computed by a single thread can dramatically affect performance. The number of threads per thread-block, the number of items computed by a thread and the "shape" of a thread-block are all specified programmatically. It is possible to combine all these configurations and optimizations in various ways, leading to a large "search space" in which to look for the best performing combination. During our tests we ran each code with several different kernel configurations in order to find the highest performance. As there are a high number of kernel configurations that are possible (e.g. legal)

we can not test them all. However, there are some rules of thumb that we adhered to when we selected the different configurations. For instance, the number of threads per thread-block should be divisible by 32, because that is the size of a *warp* in CUDA. If the number of threads per thread-block is not divisible by the warp size, it will lead to under-utilized warps. When choosing the "shape" of the thread-block, we always chose the x-dimension to be the biggest, then the y-dimension, and lastly the z-dimension. We did this because the elements in the x-dimension lies contiguously in memory, and those memory accesses can be coalesced when neighbouring threads access them, as discussed in section 2.2. In total we ended up generating and measuring the performance of about 300 different configurations during our tests. Hand-coding the same amount of configurations would be infeasible, highlighting one of the advantages by automatically generating CUDA code. Note that our tests are not exhaustive and there is a possibility that there are better performing configurations that we did not find.

We use the *nest* clause with a value of 3, to indicate that all three loop-nests should be parallelized in all test codes.

We will only test and report codes with chunking (one thread computes several elements) in the z-dimension. This is mainly because initial tests show that chunking in the x- and y- dimensions results in lower performance and partly because the Mint translator only supports chunking in the z-dimension.

We will test Spearmint with more kernel configurations than we will test the Mint translator with. The reason for that is that Mint only allows thread-blocks with a maximum size of 512 threads as that was the maximum allowed when the Mint translator was developed. Not all configurations tested on Spearmint are therefore possible with the Mint translator as Spearmint allows up to 1024 threads per thread-block. We do however test both translators with the same configurations where possible.

6.3 Baseline Performance

We first measured the performance of the Spearmint- and Mint-generated code without any command-line optimizations turned on, we will hereby

reference such code as “baseline”. We tested the translator with a nesting level of 3 and many configurations on all three test codes. Note that the Mint translator is not tested on all the tile- and chunksize configurations the Spearmint generated codes are tested on as discussed in subsection 6.2.1.

6.3.1 Baseline performance of the 3D 7-pt Heat Stencil code

The baseline performance for the 3D Heat 7-pt stencil can be seen in figure 6.1. The highest performance is measured on the codes with tile(64,8,8) and chunksize(1,1,8), resulting in 37.6 GFLOPS, for both translators. This configuration result in 512 threads per thread-block. By profiling with Nvidia's profiling tool, nvprof, we found that both the Mint- and Spearmint-generated codes achieve 91% device occupancy and they are both limited by the GPU's L2 cache's memory bandwidth.

The second best performance for Mint generated code delivers 33.5 GFLOPS with the tile(16,16,4) and chunksize(1,1,4) clauses. There are several configurations on Spearmint that delivers 36 and 35 GFLOPS, all with 1024 threads per thread-block. Mint's restriction on the thread-block sizes reduce the pool of top performing configurations some, but Mint still delivers equally good performance for the 3D 7-pt stencil with its best performing code.

Note that for all configurations that both Mint and Spearmint can generate code for, the performance differences are negligible with performance differences by less than ± 1 GFLOPS. If Mint was altered so that it could generate kernels with 1024 threads per thread-block, it is reasonable to assume that it would deliver equally good performance for such configurations as well.

Running the 3D 7-pt stencil code sequentially on one of Lizhi's Intel Xeon E5-2650 processors result in 3 GFLOPS. Utilizing the Mint model and either one of the translators result in significant performance improvements. The best performing baseline code delivers 12.5 times faster performance than the sequential code. Note that the actual speed-up will be less when including PCIe transfer costs for the GPU accelerated code that are needed before and after the computation. The only added

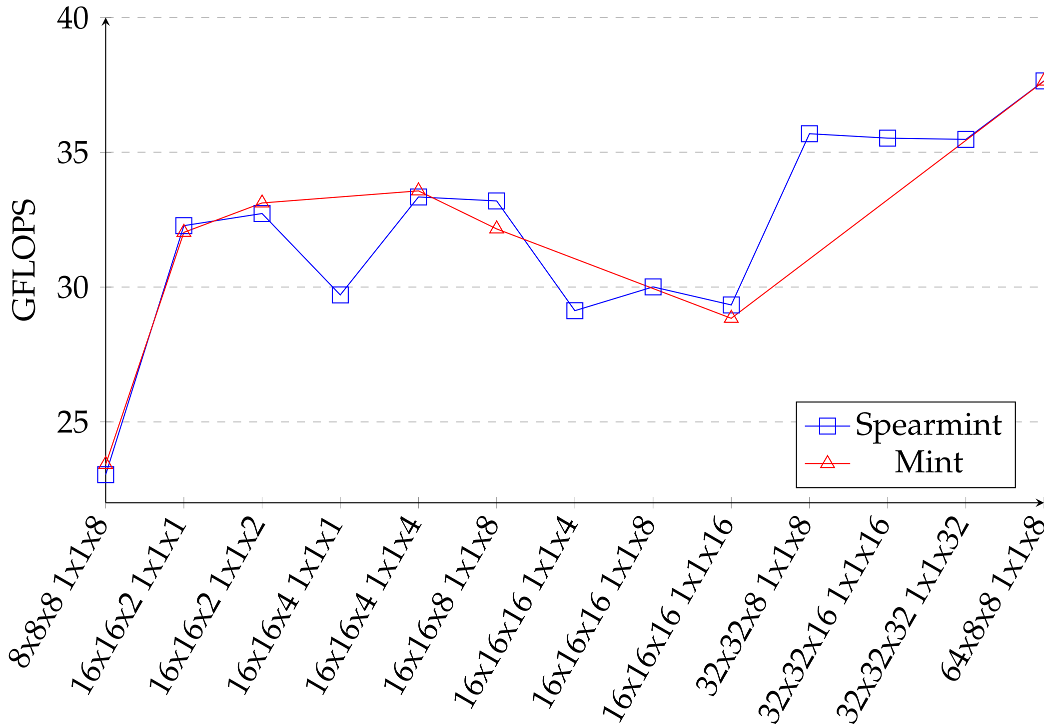

Figure 6.1: Comparison of the baseline performance of the 3D Heat 7-pt Laplace operator between Mint and Spearmint. The labels on the x-axis denotes the tile sizes and chunksizes used. For example the notation 8x8x8 1x1x8 means that the tile(8,8,8) and chunksize(1,1,8) clauses were used.

development cost by utilizing the Mint model is the addition of 6 compiler directives and implementing the specialized memory allocation that we described in section 5.3.

6.3.2 Baseline Performance of the 3D 7-pt Heat Stencil with Variable Coefficients

Figure 6.2 shows the measured baseline performance of the 3D 7-pt stencil code with variable coefficients. Also for this code Spearmint- and Mint-generated code perform equally well for the configurations that both translators allow.

The best configuration found is with the tile(64,8,8) and chunksize(1,1,8) clauses, for both translators resulting in 55.2 GFLOPS. These codes result in only 48% device occupancy, found by nvprof. The reason is that they both use 44 registers per thread, resulting in 22528 registers per

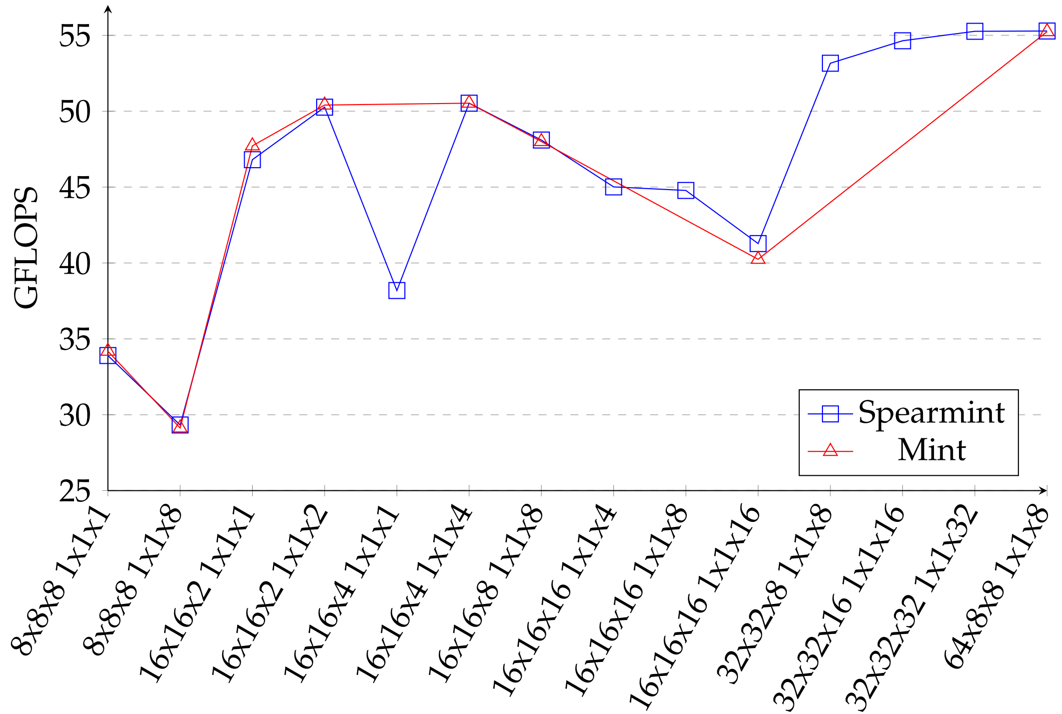

Figure 6.2: Comparison of the baseline performance of the 3D Heat 7-pt variable coefficient between Mint and Spearmint. The labels on the x-axis denotes the tile sizes and chunksizes used. For example the notation 8x8x8 1x1x1 means that the tile(8,8,8) and chunksize(1,1,1) clauses were used.

thread-block which lowers the number of parallel thread-blocks per SM to 2 (16 is maximum). The L2 cache hit rate is ~76% for both codes which result in L2 cache memory throughput of 288 GB/s.

The worst performing configuration is found with the tile(8,8,8) and chunksize(1,1,8) clauses, which delivered 34 GFLOPS. By profiling them we find that the small thread-block size of 64 threads per thread-block is reducing occupancy because each thread-block is executed using two warps. When running the maximum amount of thread-blocks per SM (16), only 32 warps are utilized, out of a possible 64 and device occupancy is then a little under 50%. In addition to having low occupancy, executing the maximum number of simultaneously executing thread-blocks per SM, lowers L2 cache hit rate to 68%, resulting in 193 GB/s of L2 cache memory bandwidth. The reason that the L2 cache hit rate is lower than the best performing configuration is presumably because that there are more “scattered” memory accesses due to the higher number of thread-blocks. This in turn results in a higher amount of unique memory accesses,

causing the L2 cache to overflow and triggering a load from global device memory.

Also for the 3D 7-pt stencil with variable coefficients code Spearmint offers a larger set of configurations that delivers the highest performance, as seen in figure 6.2 the three configurations with 1024 threads per thread-block are delivering performance close to the best performing configuration (tile(64,8,8,) and chunksize(1,1,8)). Choosing the maximum number of threads per block might make it easier for a novice programmer to find an acceptable configuration without having to test a high number of configurations.

The best configuration found (55.2 GFLOPS) is 62.3% faster than the worst configuration we found (34 GFLOPS), highlighting how crucial it is to carefully select the kernel configurations.

6.3.3 Baseline Performance of the 3D 19-pt Poisson Code

The baseline performances measured for the 19-pt Poisson code are similar to the two previous codes. Also for the 19-pt Poisson code a thread-block size of 512 threads using the tile(64,8,8) and chunksize(1,1,8) clauses result in the highest performance. The configuration result in 26 GFLOPS.

One interesting observation is that there are smaller performance variations between the different tile and chunksize configurations of the Poisson code, compared to the 7-pt Heat with variable coefficients. The best Spearmint generated code (26 GFLOPS) is only 50.2% faster than the worst Spearmint generated code (17.3 GFLOPS), while the same comparison for the 7-pt stencil with variable coefficients gives a difference of 62.3%. The reason is the 19-pt Poisson code's stencil pattern. Loading the data for the 19-pt stencil requires far more uncoalesced/strided memory accesses than the 7-pt variable code does. As discussed in section 2.2, uncoalesced memory accesses put more pressure on global device memory bandwidth. Memory access speed therefore becomes a much bigger factor and varying kernel configurations has a decreased relevance on performance.

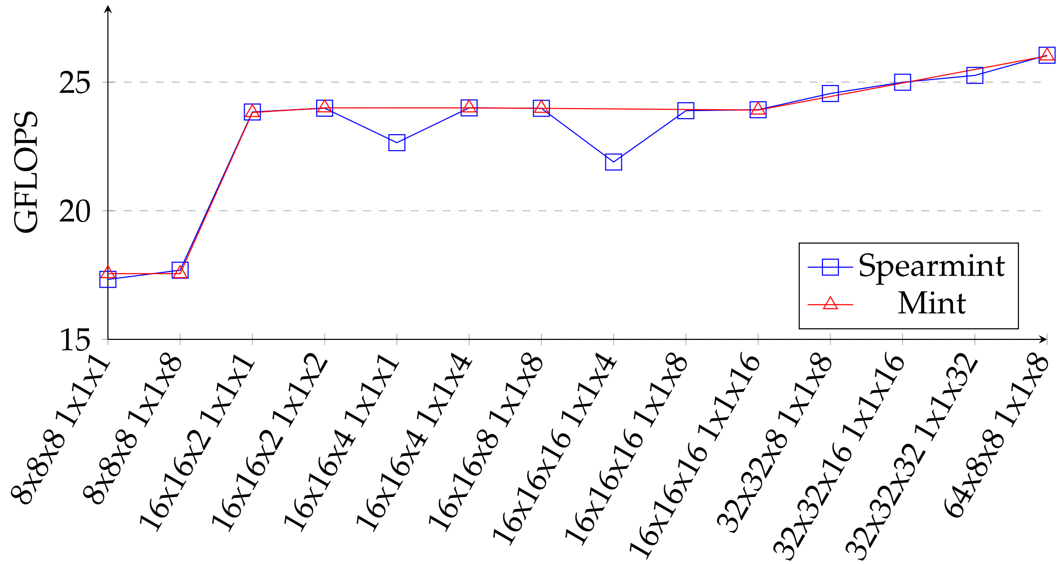

Figure 6.3: Comparison of the baseline performance of the 3D Poisson 19-pt code between Mint and Spearmint. The labels on the x-axis denotes the tile sizes and chunksizes used. For example the notation 8x8x8 1x1x8 means that the tile(8,8,8) and chunksize(1,1,8) clauses were used.

6.4 Prefer L1 Cache Configuration

We implemented a command-line configuration option that instructs the CUDA runtime to configure the L1 cache as 48KB of hardware managed L1 cache and the remaining 16KB as shared memory. This was a configuration that Didem Unat found to provide a performance benefit in certain circumstances, such as when small amounts of shared memory are used[61, p. 89]. The default configuration of the Kepler architecture favours 48KB of shared memory and 16KB of L1 cache.

The PreferL1 configuration provides a speed-up of 0-2 GFLOPS on the baseline variants of the 7-pt Heat and the 7-pt Heat with variable coefficients. On the 19-pt Poisson the baseline variant's speed-up varies from -2 GFLOPS to 1 GFLOPS. The reason for the negative performance impact this configuration sometimes incurs on the 19-pt Poisson code has been hard to pinpoint and we do not currently have any good explanation for it.

When the PreferL1 configuration is coupled with the read-only cache optimization it has no performance impact for any of the codes. The reason is that the read-only cache optimization greedily loads all memory

reads through the read-only data cache and the configuration of the L1 cache therefore becomes irrelevant.

The PreferL1 configuration might result in higher performance benefits for other codes, e.g codes that use so much register space that it results in register spilling.

6.5 Comparing Spearmint's Read-only Cache Optimization with Mint's Optimizations

In this section we compare the achieved performance by using the Mint translator's and Spearmint's optimizations. The only major optimization implemented by Spearmint is its read-only cache optimization that utilizes the Kepler architecture's read-only data cache, as described in section 5.5. All Spearmint-generated codes in this section are generated with the *-readOnlyCache* flag passed to Spearmint. The reason we only have implemented read-only cache optimization in Spearmint is due to time restrictions imposed on this project. Further optimizations should be implemented, but we leave this as future work which we will discuss in section 8.1. We did test Spearmint's preferL1 configuration but found that it does not impact performance at all while coupled with the read-only cache optimization, so we refrain from reporting it.

As described in section 3.3, the Mint translator implements shared memory optimization, register optimization and allows both optimizations to be used together. For our test-codes using Mint's register optimization alone had no effect on performance, compared to the baseline variants. The results varied by about -0.1-0.1 GFLOPS between the baseline variants and the codes with register optimization, which can be explained by system noise. We therefore do not report the results of utilizing Mint's register optimization by itself. We do however report the performance achieved when both shared memory and register optimization were used together, as that did provide performance benefits, as well as using Mint's shared memory optimization by itself. In order to generate the codes that utilize Mint's shared memory and register optimization we pass the flags *-opt:shared* and *-opt:register* to Mint. For the codes that only utilize Mint's shared memory we only pass the *-opt:shared* flag to the translator.

In addition to the restriction the Mint translator enforce on thread-block sizes discussed in subsection 6.2.1, the Mint translator also requires that the chunksize in the z-dimension is equal to the tile size in the z-dimension while using Mint's shared memory optimization[61, section 5.8]. If that is not the case, the Mint translator aborts the translation and exits with a error message. We were therefore unable to test Mint generated code with shared memory optimizations and a thread per point (chunksize of 1 in all dimensions).

6.5.1 Optimizing the 7-pt Heat Laplace Operator

Figure 6.4 shows the measured performance for the 3D 7-pt Heat stencil with optimizations turned on for both Spearmint and Mint.

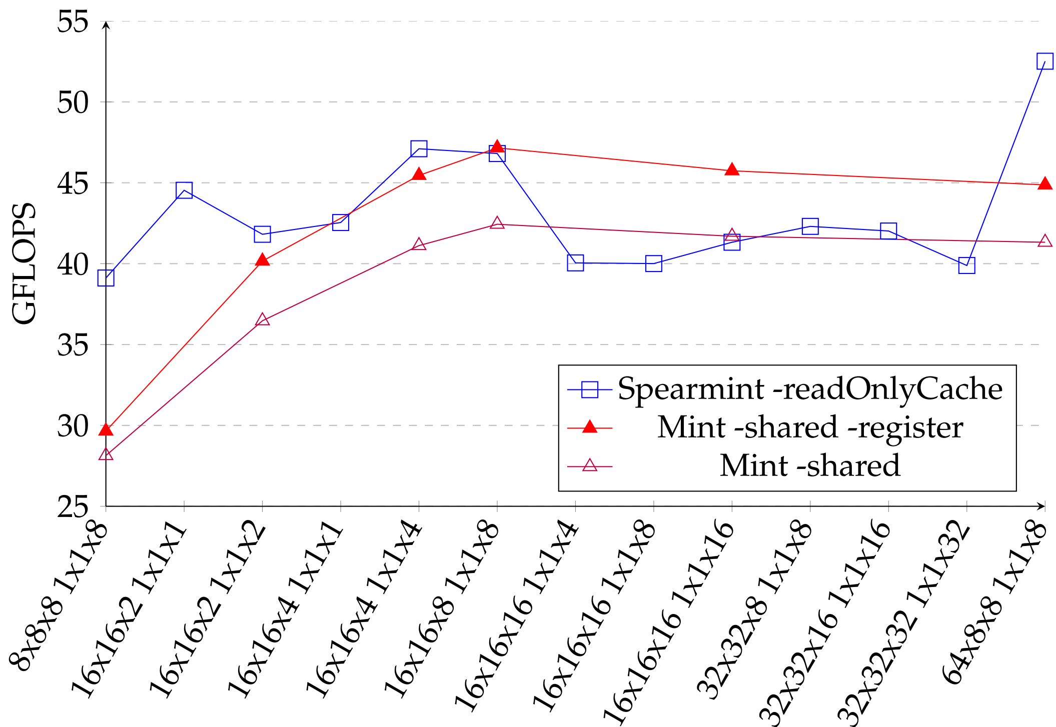

Figure 6.4: Comparison of the performance of the 3D Heat 7-pt Laplace operator between optimized Mint and Spearmint. The labels on the x-axis denotes the tile sizes and chunksizes used. For example the notation 8x8x8 1x1x8 means that the tile(8,8,8) and chunksize(1,1,8) clauses were used.

The best Spearmint-generated code delivers 52.5 GFLOPS with the tile(64,8,8) and chunksize(1,1,8) clauses. Profiling with nvprof shows that the achieved occupancy is 70.3%, with only three thread-blocks

executing simultaneously per SM. The number of thread-blocks per SM is limited by the kernels register usage. The configuration results in a read-only cache hit-rate of 65.6%, giving 370 GB/s of memory bandwidth throughput for the read-only cache. The same thread-block configuration on the Mint-generated code results in 41.3 GFLOPS with shared memory optimization and 44.8 GFLOPS with both shared memory and register optimizations. For the code with both shared memory and register optimization this configuration results in a warp execution efficiency of 74.5%, due to conditional branches in the code (if-statements). In CUDA GPUs, branching in kernel code reduces performance as the warp scheduler issues instructions for the diverging branches (different execution paths) sequentially[13, section 12.1].

The best Mint generated code uses the tile(16,16,8) and chunksize(1,1,8) clauses with both shared memory and register optimization, and results in 47.1 GFLOPS, which is 11.4% slower than the best Spearmint-generated code. This configuration results in an occupancy of 99.4%. The synchronization calls to __syncthreads() inserted by Mint are listed as the primary stall reason, meaning it is the synchronization call that usually prevents the warp from executing on any given clock cycle. Additionally, the kernel is limited by the global device memory bandwidth, with an achieved throughput of 73% of theoretical peak (208 GB/s).

With the Mint generated code it is a thread-block size of 256 and chunking of 4, 8 or 16 in the z-dimension that results in the highest performance. Both increasing and decreasing the size of the thread-block are degrading performance. For the best Spearmint generated code it is a thread-block size of 512 threads that gives the best performance. Following that, it is 256 threads per thread-block that gives the second best performance, while 1024 threads per thread-block is among the poorest performing thread-block sizes. This differs from the baseline variant where thread-blocks with 1024 threads result in the highest performance. This is caused by the fact that both Mint's optimizations and Spearmint's optimization utilize on-chip memory. On-chip memory is scarce and overuse leads to lower occupancy in the case of shared memory since the SM does not have resources enough to run the maximum number of thread-blocks in parallel.

Spearmint's read-only cache optimization greedily places as many

arrays as it can in the read-only cache, which may lead to a higher number of cache misses, compared to a optimization that tries to balance the usage, as discussed in section 5.5. As seen in figure 6.4, the configuration with tile(16,16,2) and chunksize(1,1,2) gives a performance of 41.8 GFLOPS for Spearmint, 10.4 GFLOPS less than the best performance found for Spearmint. An interesting observation for this and the best performing configuration is that they both result in roughly the same occupancy. The best performing configuration has bigger, but fewer thread-blocks (3) executing in parallel on a SM, while the poorer performing code has twice as many (6), but smaller thread-blocks per SM. Both result in an occupancy of about 70%. Since the poorer performing configuration has a higher number of thread-blocks, it is more likely that non-neighbouring thread-blocks are executing simultaneously on a SM, since we have no control over the order of which thread-blocks are executed in CUDA. This in turn can lead to a higher number of non-neighbouring points placed in the read-only cache, which increases the number of cache misses. We found that this is indeed the case for these two codes, by profiling with nvprof. The read-only cache's throughput is 292 GB/s for the poorer performing configuration, whilst it is 370 GB/s for the best performing configuration.

6.5.2 Optimizing the 3D 7-pt Heat with Variable Coefficients

The measured performances for the 3D 7-pt Heat stencil with variable coefficients are shown in figure 6.5. The best performance by Spearmint with read-only cache optimization is 71.4 GFLOPS. The best performing configuration has a tile size of 128_x, 8_y, 8_z and a chunksize of 1_x, 1_y, 8_z giving a thread-block size of 1024.

An interesting observation is that the forth-best configuration (67.2 GFLOPS) has a thread-block configuration of 4_z,4_y,4_x (64 threads) with no chunking. The same configuration without read-only cache optimization delivers one of the poorest performances observed for the baseline variant with 22.8 GFLOPS. Nvprof shows that the small thread-block sizes result in 16 active thread-blocks per SM (maximum), but since the thread-blocks are so small, less than half of the available warps per thread-block are utilized. As we have seen with the 3D 7-pt Heat stencil in subsection 6.5.1,

low occupancy can be beneficial when using Spearmint's read-only cache optimization.

The lessons learned from the baseline variants do not necessarily transfer to codes that utilize on-chip memory. Striking a balance between SM occupancy and on-chip memory usage is important. Unlike with the baseline variants, where finding the configuration that yields the highest level of occupancy usually provides good performance, it becomes more important to take the stencil computation and access pattern into account.

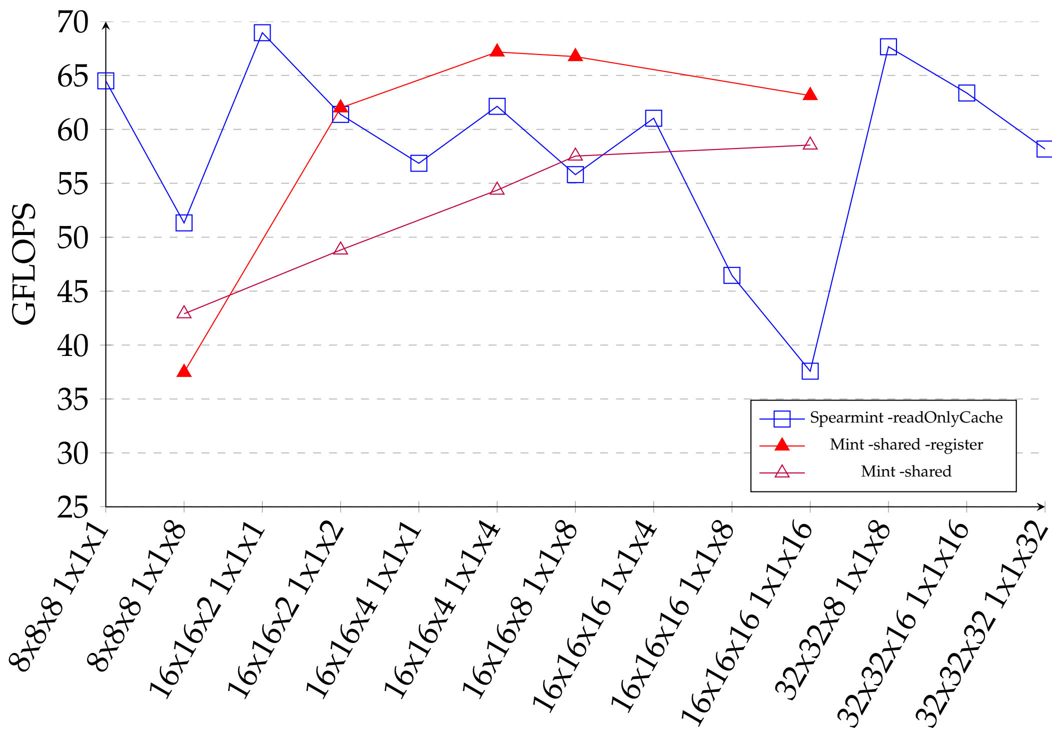

Figure 6.5: Comparison of the performance of the 3D Heat 7-pt variable coefficient between optimized Mint and Spearmint. The labels on the x-axis denotes the tile sizes and chunksizes used. For example the notation 8x8x8 1x1x1 means that the tile(8,8,8) and chunksize(1,1,1) clauses were used.

The worst performing configuration with Spearmint's read-only cache optimization has a tile size of 16 in all dimensions and a chunksize of 1_x, 1_y, 16_z, with a resulting performance of 37.5 GFLOPS. This configuration is in fact performing worse than many of the baseline configurations. By profiling with nvprof, we found that the worst performing configuration uses 36 registers per thread, resulting in 9216 threads per thread-block. Since each SM on the Tesla K20m card contains

65536 registers, 6 simultaneous thread-blocks may be run on each SM. This gives an occupancy of 70.1% per SM. The read-only cache hit rate is 61.1% with a throughput of 162.1 GB/s. The low hit rate of the read-only cache results in a higher number of L2 cache/global memory accesses, which negatively affect performance. Comparably, the best performing configuration uses 36 registers per thread as well, but because it has bigger thread-blocks (1024 threads) this results in 36864 registers for each thread-block. This allows for only one thread-block at a time per SM and lowers SM occupancy to 46.9%. The fact that only one thread-block is executed at a time increases the locality of the memory accesses placed in the read-only data cache. This gives a read-only cache hit rate of 74.9% resulting in 308GB/s of read-only cache throughput. The increase in the read-only cache throughput is 90% higher for the best performing configuration, compared to the worst. At the same time the performance difference between the best- and worst-configuration is 90.4% and it therefore seems to be a strong correlation between the utilization of the read-only data cache and the achieved performance, for this particular code.

The best performing code generated by Mint utilizes Mint's shared memory and register optimizations and uses the tile(16,16,4) and chunksize(1,1,4) clauses. It delivers a performance of 67.1 GFLOPS and is slightly slower than the best performing Spearmint generated code, with a 6.4% difference. Nvprof reveals that the limiting factors for this configuration are stalls caused by synchronization calls to __syncthreads() and the global device memory. The code achieves 71% of peak theoretical global memory bandwidth (208 GB/s).

For this code, finding a kernel configuration that delivers good performance with Spearmint needs a bit more experimentation and some configurations even perform worse than the baseline variants. Further improving the read-only cache optimization as discussed in section 5.5, might avoid generating codes that perform worse with the read-only cache optimization than without.

6.5.3 Optimizing the 3D 19-pt Poisson Code

With the 3D 19-pt Poisson code we observed the biggest differences between the Mint translator and Spearmint. The measured results are

shown in figure 6.6.

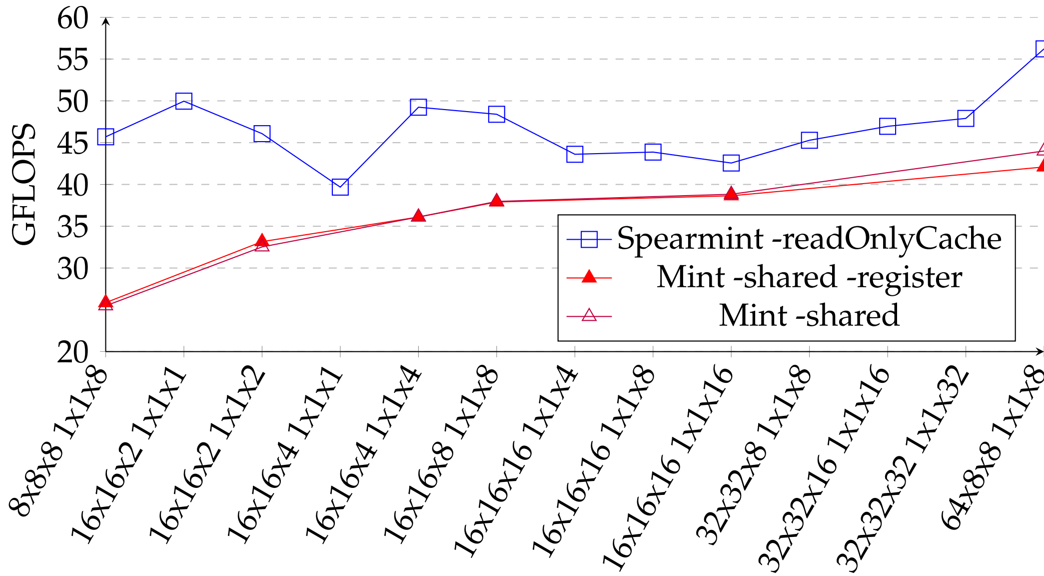

Figure 6.6: Comparison of the performance of the 3D Poisson 19-pt code between optimized Mint and Spearmint. The labels on the x-axis denotes the tile sizes and chunksizes used. For example the notation 8x8x8 1x1x8 means that the tile(8,8,8) and chunksize(1,1,8) clauses were used.

The best Spearmint configuration utilizes Spearmint's read-only cache optimization and results in a performance of 56.2 GFLOPS. The best performing Mint configuration utilizes Mint's shared memory optimization alone and results in 44 GFLOPS. The best Spearmint configuration performs 27.7% better than the best Mint configuration. Both aforementioned results are with a tile configuration of 64_x, 8_y, 8_z and a chunking of 8 in the z-dimension.

The best configuration on Spearmint results in a read-only cache throughput of 544.6 GB/s, which is the highest read-only cache throughput we have observed during our tests. The configuration also results in maximized utilization of the L2 cache bandwidth, with 416.2 GB/s. This is caused by the fact that loads from global device memory first go through the L2 cache, before they go through the read-only cache. The limiting factor for this configuration is the bandwidth of the L2 cache, as the read-only cache bandwidth is only moderately utilized.

For the Mint-generated code the best configuration results in lower achieved global device memory bandwidth (112.3 GB/s), than the Spear-

mint generated code (140.8 GB/s), as there are fewer uncoalesced accesses in the Spearmint generated code.

It seems that for codes with a high number of uncoalesced/strided memory accesses, such as the 19-pt Poisson code, Spearmint's read-only cache optimization performs better than Mint's shared memory optimization. We believe the reason is that the read-only cache allows full speed unaligned memory access patterns[42]. However, this should be further researched with large stencils.

For the 19-pt Poisson code, Mint's shared memory optimization alone performed equally well, and in one case better, than the shared memory and register optimization combined. For the 3D 7-pt Heat and the 3D 7-pt Heat with variable coefficients, combining Mint's shared memory and register optimizations generally performs better than just Mint's shared memory optimization by itself. This is because the stencils for the 3D 7-pt Heat and the 3D 7-pt Heat with variable coefficients fulfill the requirement for the additional improvement discussed in section 3.3, and the neighboring points in the z-dimension are therefore placed in registers. The 19-pt Poisson stencil does not fulfill the requirement and therefore must use more shared memory per thread-block.

6.6 Code Quality and Usability

One of the goals for this project is to create a source-to-source translator that was easy to use and that can be used as a learning tool. The context for this goal is that our target-user is a non-expert programmer whose main area of expertise is not necessarily high-performance computing.

We had some issues with the Mint translator while performing our tests. First and foremost, the Mint generated code would not compile directly, but needed some manual modifications. Some of the issues were unmatched parenthesis for method calls and references to variables that were not declared until later in the code, resulting in "undeclared variable" errors while compiling. Additionally, the forward declaration of the generated kernels had mismatching kernel signatures, which caused compiler errors as well. Writing $x <= N$ in the conditional part of a for-loop to be parallelized, instead of the semantically equivalent $x < N + 1$ resulted in CUDA code that did not compute the correct result. These

problems most definitively made the Mint generated code harder to work with as we did test a large number of different configurations and we had to hand-correct the errors in the Mint-generated codes.

In the code generated by the Mint translator, all usage of defined values (e.g. #define PI 3.1415) are swapped out with the defined value for all appearances of that definition. This seems to be an artifact caused by Mint working directly with an AST that does not retain information about how the original source looked like, but instead keeps synthesized values in its representation. In codes where defines are used extensively, this can make it harder to continue working with Mint generated code.

The Mint generated code uses a lot of unnecessary basic blocks/compound statements (e.g. { ... }) and does not insert any empty lines that logically separate the code.

Note that the problems we experienced are with the Mint version released with the Rose compiler infrastructure (version 0.9.7.20). Mint has also been made available online on [38], but requires an older browser to function properly. The Mint online editor allows the user to input and compile Mint-annotated code, and then view the generated result. After we had performed our tests we checked if the Mint online editor had the same issues. In the Mint online editor most of the issues that gave error when compiling with Nvidia's nvcc were corrected. However, the pointer swapping performed in between iterations discussed in section 6.2 still was faulty, resulting in non-functioning code. However, there might exist a Mint version that addresses all these issues that is not made available to the public or that we are not aware of.

All of the issues with the Mint translator discussed above, do not apply to Spearmint, and this makes it easier and more effective to work with Spearmint. The tests performed on Spearmint were largely automated, and were run overnight.

Baseline code generated by Mint has 51%, 70% and 39% more lines on average than the original input code for the 3D 7-pt Heat, 3D Heat with variable coefficients and the 19-pt Poisson code, respectively. For the Mint with shared memory optimizations the generated code has 68%, 92% and 62% more lines on average than the original input code for the 3D 7-pt Heat, 3D Heat with variable coefficients and the 19-pt Poisson code, respectively. The Mint-generated code with shared memory and register

optimization combined, had 68%, 95% and 67% more lines on average than the original input code, respectively.

In comparison, the Spearmint generated code for both the baseline variants and read-only cache optimized versions increased the number of lines by an average of 40%, 57% and 33% for the 3D 7-pt Heat, 3D Heat with variable coefficients and the 19-pt Poisson code, respectively. Spearmint's read-only cache optimization does not add any code lines compared to the baseline variant, it simply wraps array references that are to be placed in the read-only cache with the __ldg() intrinsic instruction. This keeps the code short and makes it easily recognizable when comparing the generated code with the input code.

6.7 Summary

In this chapter we have evaluated our source-to-source translator, Spearmint. We have compared the CUDA code generated by Spearmint with the CUDA code generated by the Mint translator. The comparisons were performed on three common three-dimensional stencil codes which are described in section 6.2.

For the baseline codes we found that the performance differences between Mint and Spearmint were negligible for the thread-block configurations they both could generate code for. Only Spearmint could generate code for the best performing baseline configurations found using 1024 threads, as the Mint translator imposes restrictions on the thread-block sizes. However, if the restriction Mint imposes on thread-block sizes was removed, it is reasonable to assume that Mint could generate comparable code for these configurations as well.

We have showed that Spearmint's read-only cache optimization delivers performance comparable to, or slightly higher than the Mint translator and its optimizations for the 3D 7-pt Heat and the 3D 7-pt Heat with variable coefficients on Nvidia's Kepler architecture. The highest performing configurations using Spearmint's read-only cache optimization results in 11.4% and 6.4% higher performance than the highest performances found with Mint's optimizations for the 3D 7-pt Heat and the 7-pt Heat with variable coefficients, respectively.

For the 19-pt Poisson stencil we found that the best Spearmint-

generated code with read-only cache optimization delivers 27.7% better performance than the best Mint-generated code with shared memory optimization. This may indicate that Spearmint's read-only cache optimization performs better than Mint's optimizations for larger stencils with a high number of uncoalesced/strided memory accesses on Nvidia's Kepler architecture. However, further research is needed to confirm this.

We have provided a discussion of the code quality and usability delivered by the two translators in section 6.6. The Mint translator does not generate code that compiles directly, but needs minor corrections in order for it to successfully compile. It also generates kernel code that is longer and more involved than Spearmint does. During our tests we experienced that working with Mint took more time than working with Spearmint, caused by the manual corrections we had to make in the Mint-generated code.

Chapter 7

Related Work

There have been many efforts to create source-to-source translators that aim to simplify parallel programming. We will not exhaust the reader with them all and only describe the ones we think are the most relevant to our work in this chapter.

The one closest related to our solution is the Mint translator, which we discussed in section 3.3. The Mint translator is intended to work with the Mint model which is described in chapter 3. Our work is largely based on, and motivated by the Mint model and the Mint translator. The Mint translator accepts Mint-annotated C code and generates optimized CUDA C code. The Mint translator was shown to deliver performance comparable to hand-optimized CUDA code for a selected set of stencil computations on CUDA's Fermi architecture[61, section 6.2]. The translator was built on the Rose compiler infrastructure[51]. Users have reported that the translator is hard to install. In addition, there are other drawbacks related to using the Mint translator, which we discussed in section 6.6.

In [53], Sourouri et al. present the Panda compiler framework. The Panda compiler framework was created for supercomputer clusters where each node consists of both CPUs and CUDA GPUs. The Panda compiler framework is directive based and offers 4 different directives to annotate serial C code with. The Panda source-to-source compiler which is also presented in [53], supports these directives. The compiler is built using the Rose compiler infrastructure[51]. The Panda compiler accepts Panda-annotated C code and generates code for homogeneous CPU clusters and heterogeneous CPU-GPU clusters. Panda can generate MPI code that runs several CPUs in parallel, MPI+CUDA code for GPU clusters

and MPI+CUDA+OpenMP for concurrent CPU+GPU execution on GPU clusters. Panda's GPU-only code was showed to scale to more than 4000 GPUs. Panda generates CUDA code that utilizes the Kepler architecture's read-only data cache. Kernel configurations such as thread-block size and chunking are passed to the generated program as arguments on run-time. This is different to the Mint model where such configurations must be specified on compile-time. A limitation of Panda is that it currently only allows stencils with up-to 7-points[53].

Noaje et al. created a translator that directly translates from OpenMP annotated C code to CUDA C in [40]. OpenMP does not have directives that inform the translator/compiler about memory movements between the host and the accelerator, and the translator must perform this task unguided. Choosing the kernel configurations is also left to the translator. This translator has chosen a higher-level abstraction than the Mint model implying the pros and cons discussed in section 2.3. It also does not support nested for-loops, resulting in a maximum nesting level of 1, to put it in Mint's terms. This is an obvious drawback when considering multi-dimensional computations.

The open-source project Par4All's translator takes sequential C code and automatically generates CUDA code, without any intervention from the programmer[47]. This can be viewed as a very high-level abstraction as the programmer has no control over the translation process. As earlier stated in section 2.3, this makes the translator's job of optimizing more difficult and the programmer's workload is not reduced much, compared to using the Mint model and its pragmas.

One source-to-source *transformer*[1] that utilizes Clang, is Scout[31]. The Scout transformer transforms Scout-annotated C code to C code where vectorized loops are augmented with SIMD intrinsics. Scout is built using Clang to generate an AST for the given source input. The vectorizations and optimizations are done by transforming the Clang-generated AST. After Scout has performed its transformations, the transformed AST is then rewritten back to C code. We considered the same approach used in Scout, in our source-to-source translator. However, Clang's AST is intended to be immutable and Clang does not facilitate transformations on the AST. Clang does have functionality for transforming the AST

[1] Transforms, meaning it transforms the AST, instead of translating it.

that Clang's semantic library use when handling C++ templates. That functionality is not offered as a part of Clang's public interface, however. Another thing that complicates AST-transformations is the AST's tight coupling to the source-input through its extensive gathering of source locations during AST-generation. Transforming the AST would often mean invalidating the source locations of the affected AST-nodes. In addition, Clang's AST does not represent comments, defines and other pragmas so information about the source is lost during the translation if one were to use the AST's pretty-printing functionality to turn the transformed AST back to source code.

Chapter 8

Future Work and Conclusion

This chapter presents our thoughts on future work and concludes the work presented in this thesis, with respect to the goals presented in section 1.1.

8.1 Future work

Future work on our source-to-source translator involves improvements to the already-implemented read-only cache optimization as well as implementing more optimizations. Future optimizations are discussed in subsection 8.1.1. Additionally, improvements that not necessarily relate to performance can be made and are discussed in 8.1.2.

8.1.1 Further Optimizations

As discussed in section 5.5, our read-only cache optimization is quite simple, and we believe that refining it could lead to performance improvements. Due to time-restrictions, further improvements are left as future work.

Shared memory- and register-optimization are still relevant for the Kepler architecture. Currently Spearmint only uses 48KB of on-chip memory in the form of read-only cache. By implementing shared memory optimization, an additional 48KB can be used. Using both the shared memory optimization and the read-only cache optimization would effectively double the amount of data in on-chip memory, compared to what it currently uses.

Nvidia's Kepler architecture introduced the shuffle intrinsic instruction. The instruction allows threads in the same warp to share register values without going through shared or global memory[section 1.4.3.3][60][42]. The shuffle instructions' latency is lower than shared shared memory accesses. The instruction does not use shared memory, so shared memory may be used for other purposes than data-sharing between threads in the same warp. As all threads in a warp performs the same instruction, there is no need for explicit synchronization, as that is provided implicitly by the warp scheduler. The shuffle intrinsic has been showed to improve performance of real-world flood simulations[24], and further research includes finding out whether this intrinsic instruction can be used in Spearmint, or not.

8.1.2 General Improvements

The source-to-source translator presented in this thesis dramatically reduces development time when experimenting with different kernel configurations. However, the programmer must manually alter the input code's *tile* and *chunksize* clauses for each configuration. Automating this process can remove that manual effort altogether. For example, the configurations can be specified as options to the translator or to the generated CUDA program in a similar way as with the Panda translator discussed in section 7. Exactly how this should be implemented remains as future work.

Future work includes investigating the possibility of moving the modifications we made to Clang into Spearmint's code base by using Clang's libPlugin library, as discussed in section 5.7. Moving the Clang modifications into Spearmint's code base would further decouple Spearmint from Clang and simplify the maintenance needed to follow Clang's development.

It also remains to implement support for the Mint model's single directive and the reduction clause to the Mint model's for directive in Spearmint.

Newer CUDA GPUs support dynamic parallelism by allowing kernel calls from within a CUDA kernel executing on the GPU. CUDA's dynamic parallelism can be used to allow nesting of the Mint model's for directive,

providing parallelization of more than three loop-nests. In order to achieve this, modifications must be made to both the Mint model and to Spearmint.

The Mint model can be further simplified by utilizing CUDA 6's unified memory[25, 64]. Memory movements between the host and the device can be handled automatically by the CUDA runtime in CUDA GPUs that supports CUDA's unified memory. Generating code that handles memory movements between the host and the device becomes functionally unnecessary, and the usage of the Mint model's copy directives may be offered as optional instead of as a requirement.

Spearmint does not generate code that checks- and communicate-errors after memory copies and kernel executions, as discussed in subsection 5.3.1. Checking and reporting errors are functionality that is nice to have because without it, a failing kernel execution for example, will occur unbeknownst to the user. The error-checking functionality should be configurable, so that the user can specify in what way errors are reported. Exactly how this functionality is configured and how it is implemented are left as future work.

8.2 Conclusion

Programming heterogeneous CPU-GPU systems is a time-consuming and complex task, especially for the non-expert programmer such as a scientist whose main field of research is not high-performance computing. Using the Mint programming model simplifies the task of defining parallelism and does not require that the programmer has detailed knowledge of the underlying hardware architecture. Code annotated with the Mint model's annotations is backwards compatible, meaning that the annotations has no effect when compiling with a regular C/C++ compiler.

The goals for this master thesis are defined in section 1.1. The major goal of this thesis was to create a source-to-source translator that translates sequential Mint-annotated C and C++ code to CUDA C/C++ code. By utilizing the LLVM compiler infrastructure we have implemented our source-to-source translator named Spearmint. Spearmint accepts sequential Mint-annotated C and C++ code and generates parallelized CUDA C/C++ code. The generated code can then be compiled with

Nvidia's nvcc[1] and executed on a system with a CUDA GPU. The translator implements one optimization that utilize CUDA GPUs' on-chip read-only data cache, for GPUs with Nvidia's Kepler architecture or newer. Additionally, optimizations such as chunking and elimination of common expressions such as index computations are implemented.

We compared Spearmint to the Mint translator on three commonly used stencil computations. We found that it delivers performance comparable to, or slightly higher than the code generated by Mint on Nvidia's Kepler architecture. For the 3D 7-pt Heat and the 3D 7-pt Heat with variable coefficients, the best Spearmint-generated code achieved 11.4% and 6.4% higher performance than Mint's best performing codes, respectively. For the 3D 19-point Poisson code the best Spearmint-generated code achieved 27% higher performance than the best Mint-generated code. Our findings indicate that Spearmint's read-only cache optimization might perform better than Mint's optimizations for larger stencils. However, further research is needed to confirm this. We think that we have achieved our goal that states that the translator should be able to generate high performance on Nvidia's CUDA GPUs based on the Kepler architecture. We have also identified potential improvements that may result in even higher performing code, as discussed in section 8.1.

We wanted the installation process of our source-to-source translator to be as simple as possible, in order to make it simple to get started using it. We provide an installation script that performs the entire installation without user intervention. The only actions the user has to take is downloading and running the script. The script has been successfully tested on Windows 10 (using Cygwin), Mac and Linux.

The translator can be used as a learning tool by examining how the input code is translated. All threads in a CUDA kernel execute the contents of the kernel code (SIMT), this is conceptually very different from regular sequential code, and by viewing how Spearmint translate a given code, it might get easier to understand the concept. Additionally, common tasks in CUDA such as moving memory between the host and the device can be viewed in the generated code. We tried to make the generated code resemble the input code as much as possible by using the variable names in the original input code and only make changes that were strictly

[1] Nvidia's CUDA C/C++ compiler

necessary. We hope that this make the generated code easy to understand and that working further with the generated code is simple.

The translator accepts a combination of Mint annotations and OpenMP annotations as a result of basing the translator on the LLVM architecture. The OpenMP annotations simply remains untouched during the translations and remains as they were, in the final generated code. However, one should not use OpenMP annotations inside any of Mint's directives, doing so result in undefined behaviour.

Bibliography

[1] Intel Corporation. URL: http://www.intel.com/content/www/us/en/homepage.html (visited on 31/10/2016) (cit. on p. 13).

[2] ACM's Software System Award, 2012. URL: http://awards.acm.org/award_winners/lattner_5074762.cfm (visited on 31/10/2016) (cit. on p. 40).

[3] *A CUDA Dynamic Parallelism Case Study: PANDA*. NVIDIA Corporation. URL: https://devblogs.nvidia.com/parallelforall/a-cuda-dynamic-parallelism-case-study-panda (visited on 31/10/2016) (cit. on p. 30).

[4] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811 (cit. on p. 49).

[5] Krste Asanovic et al. *The landscape of parallel computing research: A view from berkeley*. Tech. rep. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006 (cit. on p. 14).

[6] E. Bartocci et al. 'Toward Real-time Simulation of Cardiac Dynamics'. In: *Proceedings of the 9th International Conference on Computational Methods in Systems Biology*. CMSB '11. Paris, France: ACM, 2011, pp. 103–112. ISBN: 978-1-4503-0817-5. DOI: 10.1145/2037509.2037525. URL: http://doi.acm.org/10.1145/2037509.2037525 (cit. on p. 1).

[7] OpenMP Architecture Review Board. *OpenMP Application Programming Interface*. Version 4.5. OpenMP Architecture Review Board. 2015. URL: http://www.openmp.org/mp-documents/openmp-4.5.pdf (visited on 31/10/2016) (cit. on pp. 20, 21).

[8] *Build, Test and Package Your Software With CMake*. Kitware. URL: https://cmake.org (visited on 31/10/2016) (cit. on p. 40).

[9] Yingju Chen and Jeongkyu Lee. 'Ulcer Detection in Wireless Capsule Endoscopy Video'. In: *Proceedings of the 20th ACM International Conference on Multimedia*. MM '12. Nara, Japan: ACM, 2012, pp. 1181–1184. ISBN: 978-1-4503-1089-5. DOI: 10.1145/2393347.2396413. URL: http://doi.acm.org/10.1145/2393347.2396413 (cit. on p. 1).

[10] *clang: a C language family frontend for LLVM*. The Clang Team. URL: http://clang.llvm.org (visited on 31/10/2016) (cit. on p. 42).

[11] *Clang - Features and Goals*. The Clang Team. URL: http://clang.llvm.org/features.html (visited on 31/10/2016) (cit. on pp. 43, 44).

[12] *Clang Plugins*. The Clang Team. URL: http://clang.llvm.org/docs/ClangPlugins.html (visited on 31/10/2016) (cit. on p. 72).

[13] *CUDA C Best Practices Guide*. NVIDIA Corporation. URL: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/ (visited on 31/10/2016) (cit. on pp. 20, 26, 31, 34, 35, 64, 76, 87).

[14] *CUDA C Programming Guide*. NVIDIA Corporation. URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide/ (visited on 31/10/2016) (cit. on pp. 8, 15, 30, 33, 34, 58, 62, 63).

[15] *CUDA LLVM Compiler*. NVIDIA Corporation. URL: https://developer.nvidia.com/cuda-llvm-compiler (visited on 31/10/2016) (cit. on p. 40).

[16] Y. Cui et al. 'Physics-based Seismic Hazard Analysis on Petascale Heterogeneous Supercomputers'. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '13. Denver, Colorado: ACM, 2013, 70:1–70:12. ISBN: 978-1-4503-2378-9. DOI: 10.1145/2503210.2503300. URL: http://doi.acm.org/10.1145/2503210.2503300 (cit. on p. 1).

[17] *Cygwin*. URL: https://www.cygwin.com (visited on 31/10/2016) (cit. on p. 70).

[18] Ron Cytron et al. 'Efficiently Computing Static Single Assignment Form and the Control Dependence Graph'. In: *ACM Trans. Program. Lang. Syst.* 13.4 (Oct. 1991), pp. 451–490. ISSN: 0164-0925. DOI: 10.1145/115372.115320. URL: http://doi.acm.org/10.1145/115372.115320 (cit. on p. 41).

[19] Danny Dig et al. 'Relooper: Refactoring for Loop Parallelism in Java'. In: *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*. OOPSLA '09. Orlando, Florida, USA: ACM, 2009, pp. 793–794. ISBN: 978-1-60558-768-4. DOI: 10.1145/1639950.1640018. URL: http://doi.acm.org/10.1145/1639950.1640018 (cit. on p. 8).

[20] Jianbin Fang, Ana Lucia Varbanescu and Henk Sips. 'A Comprehensive Performance Comparison of CUDA and OpenCL'. In: *Proceedings of the 2011 International Conference on Parallel Processing*. ICPP '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 216–225. ISBN: 978-0-7695-4510-3. DOI: 10.1109/ICPP.2011.45. URL: http://dx.doi.org/10.1109/ICPP.2011.45 (cit. on p. 22).

[21] *GCC, the GNU Compiler Collection*. Free Software Foundation, inc. URL: https://gcc.gnu.org (visited on 31/10/2016) (cit. on p. 36).

[22] *Getting Started with the LLVM System. Requirements*. LLVM Project. URL: http://llvm.org/docs/GettingStarted.html#requirements (visited on 31/10/2016) (cit. on pp. 40, 71).

[23] LLVM Developer Group. *The LLVM Compiler Infrastructure*. URL: http://llvm.org (visited on 31/10/2016) (cit. on pp. 2, 40).

[24] Zsolt Horváth et al. 'Kepler shuffle for real-world flood simulations on GPUs'. In: *International Journal of High Performance Computing Applications* (2016). DOI: 10.1177/1094342016630800 (cit. on pp. 73, 102).

[25] *Inside Pascal: NVIDIA's Newest Computing Platform*. NVIDIA Corporation. URL: https://devblogs.nvidia.com/parallelforall/inside-pascal/ (visited on 31/10/2016) (cit. on p. 103).

[26] *Intel® Hyper-Threading Technology*. Intel Corporation. URL: http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html (visited on 31/10/2016) (cit. on p. 14).

[27] *Intel® Many Integrated Core Architecture - Advanced*. Intel Corporation. URL: http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html (visited on 31/10/2016) (cit. on p. 13).

[28] *Intel® Xeon Phi™ Coprocessor Peak Theoretical Maximums*. Intel Corporation. URL: http://www.intel.com/content/www/us/en/benchmarks/server/xeon-phi/xeon-phi-theoretical-maximums.html (visited on 31/10/2016) (cit. on p. 14).

[29] *Intel's 50-core champion: In-depth on Xeon Phi*. Ziff Davis, LLC. URL: http://www.extremetech.com/extreme/133541-intels-64-core-champion-in-depth-on-xeon-phi (visited on 31/10/2016) (cit. on p. 13).

[30] Kazuhiko Komatsu et al. 'Evaluating performance and portability of OpenCL programs'. In: *The fifth international workshop on automatic performance tuning*. Vol. 66. 2010 (cit. on p. 22).

[31] Olaf Krzikalla et al. 'Euro-Par 2011: Parallel Processing Workshops: CCPI, CGWS, HeteroPar, HiBB, HPCVirt, HPPC, HPSS, MDGS, ProPer, Resilience, UCHPC, VHPC, Bordeaux, France, August 29 – September 2, 2011, Revised Selected Papers, Part II'. In: ed. by Michael Alexander et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. Chap. Scout: A Source-to-Source Transformator for SIMD-Optimizations, pp. 137–145. ISBN: 978-3-642-29740-3. DOI: 10.1007/978-3-642-29740-3_17. URL: http://dx.doi.org/10.1007/978-3-642-29740-3_17 (cit. on p. 98).

[32] Chris Lattner. In: Amy Brown. *The Architecture of Open Source Applications*. lulu.com, 2012. Chap. 11. LLVM. ISBN: 978-1257638017. URL: www.aosabook.org/en/llvm.html (cit. on p. 40).

[33] Chris Lattner and Jianzhou Zhao. Chris Lattner on the name LLVM on LLVM dev mailing list. URL: http://archive.is/T1qik (visited on 31/10/2016) (cit. on p. 40).

[34] *LibTooling*. The Clang Team. URL: http://clang.llvm.org/docs/LibTooling.html (visited on 31/10/2016) (cit. on p. 69).

[35] *LLVM Language Reference Manual. Introduction*. LLVM Project. URL: http://llvm.org/docs/LangRef.html (visited on 31/10/2016) (cit. on p. 41).

[36] Naoya Maruyama and Takayuki Aoki. 'Optimizing stencil computations for NVIDIA Kepler GPUs'. In: *Proceedings of the 1st International Workshop on High-Performance Stencil Computations, Vienna*. 2014, pp. 89–95 (cit. on p. 12).

[37] Naoya Maruyama et al. 'Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-scale GPU-accelerated Supercomputers'. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '11. Seattle, Washington: ACM, 2011, 11:1–11:12. ISBN: 978-1-4503-0771-0. DOI: 10.1145/2063384.2063398. URL: http://doi.acm.org/10.1145/2063384.2063398 (cit. on p. 6).

[38] *Mint Online Editor*. Note: requires an older browser. URL: http://mint.simula.no (visited on 31/10/2016) (cit. on p. 93).

[39] Gordon E. Moore. 'Readings in Computer Architecture'. In: ed. by Mark D. Hill, Norman P. Jouppi and Gurindar S. Sohi. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000. Chap. Cramming More Components Onto Integrated Circuits, pp. 56–59. ISBN: 1-55860-539-8. URL: http://dl.acm.org/citation.cfm?id=333067.333074 (cit. on p. 5).

[40] G. Noaje, C. Jaillet and M. Krajecki. 'Source-to-Source Code Translator: OpenMP C to CUDA'. In: *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*. Sept. 2011, pp. 512–519. DOI: 10.1109/HPCC.2011.73 (cit. on p. 98).

[41] *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. whitepaper. Version 1.1. NVIDIA Corporation, 2009. URL: https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf (visited on 31/10/2016) (cit. on p. 36).

[42] *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*. whitepaper. NVIDIA Corporation, 2012. URL: https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf (visited on 31/10/2016) (cit. on pp. 60, 92, 102).

[43] *OpenACC - Directives for Accelerators*. OpenACC-standard.org. URL: http://www.openacc.org/ (visited on 31/10/2016) (cit. on p. 21).

[44] OpenACC-standard.org. *The OpenACC Application Programming Interface*. Version 2.5. OpenACC-standard.org. 2015. URL: http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf (visited on 31/10/2016) (cit. on pp. 8, 21).

[45] *OpenMP Compilers*. URL: http://openmp.org/wp/openmp-compilers (visited on 31/10/2016) (cit. on p. 20).

[46] *OpenMP - The OpenMP® API specification for parallel programming*. URL: http://openmp.org (visited on 31/10/2016) (cit. on p. 20).

[47] *Par4All: Auto-Parallelizing C and Fortran for the CUDA Architecture*. URL: http://pips4u.org/par4all/presentations/par4all-auto-parallelizing-c-and-fortran-for-the-cuda-architecture (visited on 31/10/2016) (cit. on p. 98).

[48] *PCI Express*. Wikimedia Foundation, inc. URL: https://en.wikipedia.org/wiki/PCI_Express (visited on 31/10/2016) (cit. on pp. 26, 63).

[49] *Reduced instruction set computing*. Wikimedia Foundation, inc. URL: http://en.wikipedia.org/wiki/Reduced_instruction_set_computing (visited on 31/10/2016) (cit. on p. 41).

[50] David P. Rodgers. 'Improvements in Multiprocessor System Design'. In: *Proceedings of the 12th Annual International Symposium on Computer Architecture*. ISCA '85. Boston, Massachusetts, USA: IEEE Computer Society Press, 1985, pp. 225–231. ISBN: 0-8186-0634-7. URL: http://dl.acm.org/citation.cfm?id=327010.327215 (cit. on p. 7).

[51] *Rose*. URL: http://www.rosecompiler.org (visited on 10/10/2016) (cit. on pp. 2, 36, 39, 97).

[52] Mohammed Sourouri. 'Scalable Heterogeneous Supercomputing: Programming Methodologies and Automated Code Generation'. PhD thesis. University of Oslo, 2015. URL: http://urn.nb.no/URN:NBN:no-53942 (visited on 31/10/2016) (cit. on p. 6).

[53] Mohammed Sourouri, Scott B. Baden and Xing Cai. 'Panda: A Compiler Framework for Concurrent CPU+GPU Execution of 3D Stencil Computations on GPU-accelerated Supercomputers'. In: *International Journal of Parallel Programming* (2016), pp. 1–19. ISSN: 1573-7640. DOI: 10.1007/s10766-016-0454-1. URL: http://dx.doi.org/10.1007/s10766-016-0454-1 (cit. on pp. 6, 97, 98).

[54] *Swift. A modern programming language that is safe, fast, and interactive. Fast and Powerful*. Apple Inc. URL: https://developer.apple.com/swift/ (visited on 31/10/2016) (cit. on p. 40).

[55] LLVM Team. *Clang 3.8 Release Notes*. The Clang Team. URL: http://llvm.org/releases/3.8.0/tools/clang/docs/ReleaseNotes.html#cuda-support-in-clang (visited on 31/10/2016) (cit. on p. 44).

[56] The LLVM Team. *CommandLine 2.0 Library Manual*. LLVM Project. URL: http://llvm.org/docs/CommandLine.html (visited on 31/05/2016) (cit. on p. 44).

[57] *TESLA GPU Accelerators for workstations*. NVIDIA Corporation. URL: http://www.nvidia.com/object/tesla-workstations.html (visited on 31/10/2016) (cit. on p. 6).

[58] *The open standard for parallel programming of heterogeneous systems*. Khronos Group. URL: https://www.khronos.org/opencl/ (visited on 31/10/2016) (cit. on p. 22).

[59] *Top 500 - The List*. TOP500.org. URL: https://www.top500.org (visited on 31/10/2016) (cit. on pp. 5, 6).

[60] *Tuning CUDA Applications for Kepler*. NVIDIA Corporation. URL: http://docs.nvidia.com/cuda/kepler-tuning-guide/ (visited on 31/10/2016) (cit. on p. 102).

[61] Didem Unat. 'Domain-specific Translator and Optimizer for Massive On-chip Parallelism'. PhD thesis. La Jolla, CA, USA, 2012. ISBN: 978-1-267-25462-7. URL: https://www.simula.no/publications/domain-specific-translator-and-optimizer-massive-chip-parallelism (cit. on pp. iii, 2, 5, 25, 26, 27, 28, 32, 34, 35, 36, 39, 56, 75, 76, 84, 86, 97).

[62] 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. Tucson, Arizona, USA: ACM, 2011, pp. 214–224. ISBN: 978-1-4503-0102-2. DOI: 10.1145/1995896.1995932 (cit. on p. 34).

[63] D. Unat et al. 'Accelerating a 3D Finite-Difference Earthquake Simulation with a C-to-CUDA Translator'. In: *Computing in Science Engineering* 14.3 (May 2012), pp. 48–59. DOI: 10.1109/MCSE.2012.44 (cit. on pp. 1, 34, 36).

[64] *Unified Memory in CUDA 6*. NVIDIA Corporation. URL: https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6 (visited on 31/10/2016) (cit. on p. 103).

[65] *WHAT IS CUDA?* NVIDIA Corporation. URL: http://www.nvidia.com/object/cuda_home_new.html (visited on 31/10/2016) (cit. on pp. 10, 18).

[66] Samuel Williams, Andrew Waterman and David Patterson. 'Roofline: An Insightful Visual Performance Model for Multicore Architectures'. In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785. URL: http://doi.acm.org/10.1145/1498765.1498785 (cit. on p. 14).