ImageCL 3D Extensions Targeting Adaptive Mesh Refinement Proxy Applications on GPUs

Even Olsson Rogstadkjærnet

Master of Science in Computer Science

Submission date:	August 2018
Supervisor:	Anne Cathrine Elster, IDI
Co-supervisor:	Jan Christian Meyer, IDI

Norwegian University of Science and Technology
Department of Computer Science

Problem Description

This project will build on the ImageCL work done in the fall, extending the ImageCL language and compiler with functionality guided by proxy applications, i.e. mini-applications with performance characteristics of larger applications that can be used for performance modeling. The project will evaluate one or more of the proxy applications worked on by the Mantevo Project and develop it/them or a new application further to take advantage of the GPU using ImageCL.

Sammendrag

Mens adopsjonen av parallelle og heterogene systemer fortsetter, øker også programmerings-kompleksiteten i disse systemene. Rammeverk som Compute Unified Device Architecture (CUDA) og Open Computing Language (OpenCL) gir funksjonell portabilitet på tvers av støttede enheter. Men de er ikke i stand til å kjøre den samme koden optimalt over forskjellige arkitekturer, eller gi mulighet for å porte koden enkelt og effektivt til andre Graphics Processing Unit (GPU) arkitekturer. Denne utfordringen er kjent som ytelses portabilitet, og er viktig siden hardware arkitekturen for GPU-er kan variere mye mellom forskjellige enheter.

Ved å gjøre optimaliseringer om til tuning parametere som kan bli gjort automatisk av kompilatoren, kan en auto-tuner bli brukt til å velge den beste kombinasjonen av optimaliseringer for hver arkitektur. Denne strategien har tidligere blitt utforsket ved å bruke ImageCL språket, som abstraherer bort mye av kompleksiteten ved å gjøre mange manuelle optimaliseringer om til tuning parametere.

I denne oppgaven utvider vi ImageCL språket og kompilatoren for å kunne støtte et bredere spekter av applikasjoner. Disse utvidelsene vil bli styrt av miniAMR, en proxy applikasjon med ytelse-karakteristikken til en Adaptive Mesh Refinement (AMR) applikasjon. I tillegg genererer vi flere GPU stensil-kerneler fra ImageCL kode og integrerer dem i miniAMR applikasjonen, og klarer å få en betydelig ytelsesforbedring (opptil 6.78x) for kernelene i forhold til referanse-implementasjonen.

Abstract

As the adoption of parallel and heterogeneous systems increases, programming such systems also becomes increasingly complex. Frameworks like Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL) provides functional portability across their supported devices. However, having the same code run optimally across multiple devices with different architectures, including being able to port code fairly seamlessly and efficiently to other GPU device architectures, is not provided. This challenge, known as performance portability, is significant since Graphics Processing Unit (GPU) architectures tend to get updated and vary even more than Central Processing Unit (CPU) architectures.

By transforming optimizations into tuning parameters that can be applied statically by the compiler, an auto-tuner can be used to pick the best combination of optimizations for each architecture. This strategy has earlier been explored using the ImageCL language and compiler, which moves much of the complexity away from the programmer by abstracting away many optimizations which would normally have to be applied manually.

In this thesis, we extend the ImageCL language and compiler to support a broader range of applications. These extensions will be guided by miniAMR, a proxy application with the performance characteristics of an Adaptive Mesh Refinement (AMR) application. AMR is a computational method used for adapting the accuracy within certain regions of a domain, and is often used in scientific and engineering applications. We generate multiple GPU stencil kernels from ImageCL code and integrate them into the miniAMR application. We are able to show a considerable speedup (up to 6.78*x*) for many of the generated stencil kernels in miniAMR compared to the reference implementation.

Acknowledgements

The work on this thesis has been done under the supervision of Dr. Anne C. Elster and Dr. Jan Christian Meyer, which I would like to thank for the help and guidance given during this thesis. Thanks to NTNU for supporting the HPC-lab and providing access to GPUs, and thanks to NTNU's HPC Center for access to the EPIC cluster. I would also like to thank my fellow students at the HPC Lab for making the time spent at the lab much more enjoyable. Finally, I would like to thank my family for the support and encouragement provided during this thesis and also throughout my years of study. Thank you.

[illegible]

Even Olsson Rogstadkjærnet

Table of Contents

List of Tables

List of Figures

Listings

List of Abbreviations

ALU	Arithmetic Logic Unit
AMR	Adaptive Mesh Refinement
API	Application Programming Interface
AST	Abstract Syntax Tree
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
FFT	Fast Fourier Transform
FLOPS	Floating Point Operations Per Second
GPU	Graphics Processing Unit
HIP	Heterogeneous-compute Interface for Portability
HPC	High Performance Computing
ILP	Instruction Level Parallelism
MIMD	Multiple Instructions Multiple Data
ML	Machine Learning
MPI	Message Passing Interface
OpenCL	Open Computing Language
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Threads

SM	Streaming Multiprocessor
SPMD	Single Program Multiple Data

Chapter 1

Introduction

Since the end of Dennard scaling [9] around 2006, we have seen a shift from high frequency single-core systems to multi/many-core and heterogeneous systems. Multi/many-core systems takes advantage of parallelism to increase the performance of an application. This usually comes at the cost of the serial performance, as each core often comes with a lower frequency than in a single-core system [11]. Heterogeneous systems are able to provide both high single threaded performance and high parallel performance by combining a low latency CPU with a high throughput accelerator.

Although originally targeting graphics in computer games, GPUs provide a high level of parallelism and high memory bandwidth, which makes it well suited for data level parallelism. The GPUs thus have become some of the most popular accelerators used in heterogeneous systems, and can be found in both consumer and High Performance Computing (HPC) systems today.

The ability to get optimal performance of the same kernel across multiple devices, also known as *performance portability*, is a known problem for accelerators [46, 24, 43]. Frameworks such as CUDA and OpenCL provides functional portability across their supported devices. However, due to the wide range of available optimizations, devices, and architectural differences, these frameworks are not able to provide performance portability.

Auto-tuning has been proposed as a solution to the performance portability issue. This involves automatically generating multiple different implementations from the same code and evaluation the performance of each version. The version with the greatest performance is then chosen as the final implementation. This can either be done with exhaustive search over the domain, which can be time-consuming, or by using a model driven approach that takes advantage of heuristics to find a subset of the domain to evaluate [14, 28].

ImageCL is a language designed by Falch and Elster (2016) [12], to provide performance portability and make it easier to write image and stencil based kernels for heterogeneous hardware. ImageCL abstracts away optimization features that would normally be required to do manually, *e.g.* the memory levels. These optimizations are instead turned into tuning parameters, which can be used to train a machine learning model that can be used as an auto-tuning model. This model can in turn be used to generate optimized device code based on the platform and device it compiles for [12, 15].

The ImageCL language provides the built-in `Image` type, which represents a two-dimensional grid of pixels. The `Image` type makes it possible to adapt stencils and other common image processing code that works on two-dimensional problem domains, without restructuring the code. However, calculations that works on three-dimensional problem domains are often found in scientific and engineering code, and these are not easily adapted to the ImageCL language.

Earlier work on ImageCL includes a CUDA backend to ImageCL, and was added as part of the specialization project in the fall of 2017 [44]. This allows ImageCL to generate optimized CUDA code in addition to OpenCL code. In this thesis we will use this CUDA backend to generate our kernels from the ImageCL code.

1.1 Project Goal

In this thesis, we aim to extend the ImageCL language and its compiler to support a broader range of applications. We have decided to target miniAMR [51], an Adaptive Mesh Refinement (AMR) proxy application, because it contains a stencil computation which is well suited for ImageCL and AMR is often found in scientific and engineering applications. The miniAMR proxy application is used to guide the extensions we elect to make to ImageCL so that it can be use for a wider range of scientific applications. Our work will include investigating how well our code generated by the extensions to ImageCL is able to integrate with and improve the performance of the proxy application and its stencil kernel.

1.2 Contributions

The contributions made in this thesis includes:

- The `Block` type extension to ImageCL, for representing three-dimensional data.
- The `idz` keyword extension to ImageCL, for finding the thread identity in the z-dimension.
- The `none` boundary condition in ImageCL, which allows for manual boundary handling.
- Extensions to the data analysis applied by ImageCL to support three-dimensional optimizations.
- The offset operator in ImageCL, which allows for aliasing of `Image` or `Block` data with offset.
- The support for memory optimizations on the `Block` type. This includes shared memory, constant memory, and texture memory.
- Benchmarks of multiple miniAMR stencil kernels generated by ImageCL, including the ImageCL kernel code.

1.3 Structure of the Thesis

The remainder of this thesis is structured as follows:

Chapter 2 presents the necessary background material and information, focusing on the GPU architecture, performance portability, and proxy applications.

Chapter 3 presents the extensions added to the ImageCL language and compiler.

Chapter 4 describes how the code generated from ImageCL has been integrated into the miniAMR proxy application.

Chapter 5 presents and discusses the performance measurements of the miniAMR kernels generated by ImageCL.

Chapter 6 concludes the thesis and suggests areas suitable for future work.

Appendix A shows the full code of the ImageCL kernels that is used to generate the GPU kernels.

Appendix B illustrates the miniAMR configurations used in the performance measurements.

Appendix C presents the performance measurements in detail.

Chapter 2

Background

This chapter highlights some of the necessary background information in which the rest of the chapters are built on. Part of this chapter is based on the background chapter of the specialization project [44] of the author in the fall of 2017, where we extended the ImageCL compiler with a CUDA backend in addition to the register caching optimization for the CUDA backend. Section 2.1 gives a brief overview of different types of parallel computing, before Section 2.2 gives a detailed description of the GPU architecture and programming model. Section 2.3 explains the concept of performance portability and auto-tuning. Next, Section 2.4 describes the ImageCL compiler and language used to generate optimized GPU code from tuning parameters. Section 2.5 gives an overview of proxy applications and their use cases. Finally, Section 2.6 will briefly discuss earlier related work.

2.1 Parallel Computing

Present hardware is built to take advantage of parallelism. This requires developers to build software that is able to take advantage of this parallelism to achieve good performance. Traditionally, developers have used a serial model when designing and implementing software, but as the serial performance of hardware have stagnated, taking advantage of this parallelism have become increasingly important [11, 5, 27].

Hardware and compiler architects have tried to exploit Instruction Level Parallelism (ILP) which allows the CPU to execute multiple instructions in parallel, while still letting the user program using a serial model. However, it has shown to be difficult to take advance of ILP much further than we currently are, and ILP has reached a point of diminishing returns. Instead, more independent cores are added to the CPU that can be used to run multiple parts of the program in parallel [21].

While multi-core systems do not have the same restrictions as ILP when it comes to performance, it moves much of the complexity of parallelism on to the developers. Developers now have to represent the system using a parallel model instead of a serial model, giving more control to the developer at the cost of increasing the complexity of the software development.

2.1.1 Classification of Parallelism

Parallelism is often classified as either being task-level based or data-level based. Task-level based parallelism is the simultaneous execution of individual tasks, *e.g.* execution of two independent functions in parallel. This means that separate instruction streams work on separate data streams in parallel, also known as Multiple Instructions Multiple Data (MIMD). While it is often easy to apply this kind of parallelism on separate tasks in the application, it is usually hard to find a large portion of the application that is well suited for this kind of parallelism since tasks are often dependent on each other. MIMD can also be applied as data parallelism, *e.g.* letting each thread work on on their own separate slice of an array [21].

Data level based parallelism, or data parallelism, is the simultaneous execution of an operation on chunks of data of the same workload, *e.g.* by letting each processing element work on its own part of an array. Single Instruction Multiple Data (SIMD) is one form of data parallelism which performs the same instruction on multiple items in parallel. The SIMD parallelism is exposed as special instructions that works on multiple elements, and provides a simple way for the programmer to take advantage of parallelism while still being able to use a serial model when implementing the software. However, as all elements being operated on needs to take the same instruction-path, the SIMD model can be limited in its functionality [21].

Single Program Multiple Data (SPMD) is a specialization of MIMD where the same program is executed on multiple processors with possibly different input data. This allows the control flow for each task to diverge, rather than being executed in lockstep as with the SIMD model. However, unlike SIMD, SPMD forces the programmer to use a parallel model when building the application, which increases the complexity. Message Passing Interface (MPI) [32, 48] is a standard that can be used to gain SPMD parallelism within an application. Recently, the Single Instruction Multiple Threads (SIMT) model has become popular as it is the dominant parallel model on the GPU. SIMT is a combination of SIMD and multithreading, and will be explained in more details in Section 2.2.3 [21, 42].

2.1.2 GPU Computing

The GPU was first introduced as a fixed pipeline accelerator for graphics that was meant to offload the compute heavy workload associated with the three-dimensional graphics rendering from the CPU [42]. However, as the GPU is particularly good at highly parallel and compute intensive workloads, it has now evolved into a more general highly parallel programmable processor. The result of this evolution is a processor like NVIDIA's GV100, based on the Volta architecture, with 5120 32-bits floating point cores, 2560 64-bits floating point cores, and a peak tera-Floating Point Operations Per Second (FLOPS) count of 15.7, which is a substantially greater amount than any CPU is able to provide. In addition, the Volta architecture introduces tensor cores that are designed specifically for deep learning [29].

The GPU is designed for a particular class of applications that provides certain characteristics that the GPU is able to take advantage of. As explained in Owens et al. (2008) [42], it needs to be an application where throughput is more important than latency, the computational requirements are large, and the opportunity for data-level parallelism is substantial. These requirements is a good fit for many scientific and high performance applications, as well as games and other graphics intensive applications [42]. Today it is common to use a heterogeneous system that combines the low-latency attribute of the CPU with the high-throughput attribute of the GPU. This provides a high value of performance, price and power usage.

The major GPU vendors, like NVIDIA, AMD, and Intel, have seen the potential of general purpose computing on the GPU and released vendor specific Application Programming Interfaces (APIs) like NVIDIA's CUDA platform [37] and AMD's Heterogeneous-compute Interface for Portability (HIP) [2], or supported open standards like OpenCL [26]. OpenCL provides functional portability to all devices that supports the OpenCL standard. This includes the majority of CPUs and GPUs, and many other specialized accelerators.

2.2 The GPU Architecture

This section describes the architecture of the GPU, and we will use the CUDA terms when explaining different components found in the GPU. However, you will find many of the same concepts in other vendors architecture under potentially different names, and many of the differences between CUDA and OpenCL will be explained in detail in Section 2.2.5. NVIDIA introduced the CUDA architecture in 2007 as a parallel computing platform and programming model for general computing on the GPU [38, 37].

Figure 2.1 illustrates the main structural difference between the CPU and the GPU. We can see that the CPU provides large caches and more advanced control units, while the GPU contains many simpler Arithmetic Logic Units (ALUs). This difference in architecture is the main reason the CPU excels at low-latency workloads, while the GPU excels at compute intensive, data-parallel workloads [35, 27]. In the follow sections, we will go into details of how the GPU architecture works.

2.2.1 Thread Blocks

A thread block is an abstraction which represents a grouping of primitive threads that executes the same kernel. The number of threads within a

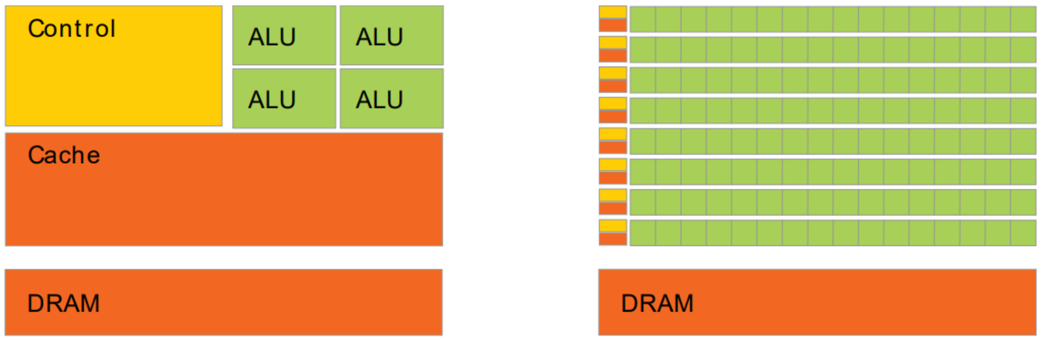


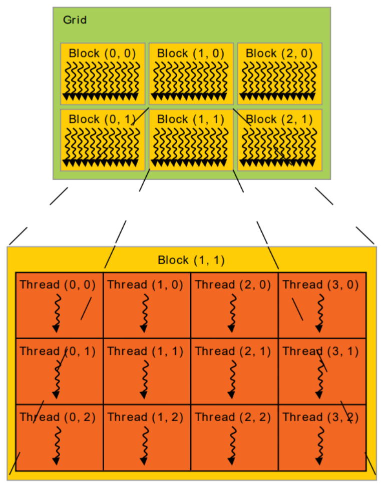
Figure 2.1: The difference in hardware on the CPU and on the GPU, focusing on the distribution of DRAM, cache, control units, and ALUs for each device. From *CUDA C Programming Guide* [35] with permission.

thread block is chosen based on the amount of resources desired by each thread and the desired occupancy on the Streaming Multiprocessor (SM). The threads can be organized in either a one-, two-, or three-dimensional layout within a single thread block, and the thread blocks are then organized into a grid in either a one-, two-, or three-dimensional layout [35, 21]. This is illustrated in Figure 2.2(a) where threads in a two-dimensional grid of two-dimensional blocks are organized.

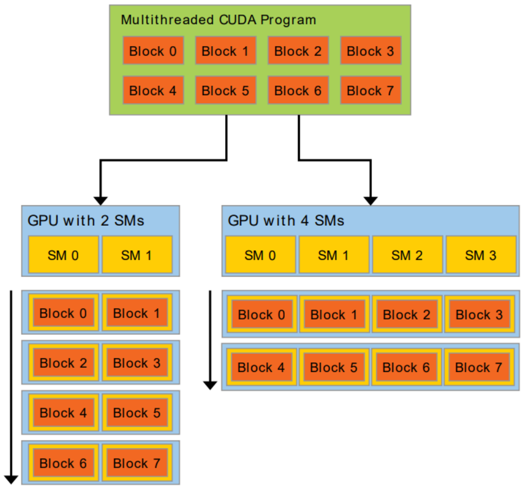
This flexibility makes it easier to work with problems like multi-dimensional matrices. In addition, the thread blocks are executed independently of each other, which allows the thread blocks to be distributed across SMs differently depending on the amount of SMs available on the GPU. As illustrated in Figure 2.2(b), the GPU can distribute the workload across all available SMs, making the program scale across different devices.

2.2.2 Streaming Multiprocessors

The SMs are the hardware on the GPU that is able to execute the users GPU kernels. Each SM contains their own registers, caches, warp schedulers, and dispatch units, as well as execution cores for integers, and 32-bits and 64-bits floating point operations [36, 27]. In addition, the Volta architecture introduced the Tensor Core, which is a mixed-precision 16-bits and 32-bits execution core built for deep learning matrix arithmetic. The warp



(a) A two-dimensional grid of two-dimensional blocks.



(b) Distribution of blocks on two devices with different amount of SMs available.

Figure 2.2: Block layout and distribution. From *CUDA C Programming Guide* [35] with permission.

scheduler and dispatch unit is responsible to schedule and switch contexts between warps, and to issue instructions to warps that are ready to execute, respectively. The Volta GV100 GPU from NVIDIA has 80 SMs that can execute blocks independently [29]. As seen in Figure 2.2(b), the thread blocks are mapped to the hardware by dividing them between available SMs. This asserts that each thread block gets the most resources available on each SM.

2.2.3 Single Instruction Multiple Threads

The SM is able to execute hundreds of threads concurrently by taking advantage of SIMT. SIMT is an execution model where multiple threads are grouped together and executes the same instructions in lockstep on different data. In CUDA, these groups are called warps and consists of 32 threads. When a SM is given a block for execution, it partitions the threads in the block into warps that is scheduled by a warp scheduler [36].

Threads within a warp have their own instruction address counter and register state which enables them to take different paths in the program. If threads within a warp take different paths, the warp executes each path while masking out threads not on the current path. This is called branch divergence and while this does not affect the correctness of the program, it causes the execution of branches to be serialized, leading to performance decrease [36].

Volta [29] introduced *Independent Thread Scheduling* which allows full concurrency between threads, regardless of warp. A *Schedule Optimizer* now instead groups active threads within a warp together into SIMT units, which enables the high throughput of earlier NVIDIA architectures in addition to increased flexibility as threads can diverge and reconverge at sub-warp granularity [36].

2.2.4 The GPU Memory Hierarchy

There are two main groups of memory: memory located on-chip and memory located off-chip in DRAM. Memory located on-chip includes registers, shared memory, and cache. Registers are the fastest memory on the GPU and is local to each thread. The maximum amount of registers per thread in Volta is 255. However, this number will be limited by the occupancy on the SMs, and with a 100% occupancy each thread will only have access to 32 registers [29]. Shared memory is a user managed cache that is shared between all threads in the same thread block. Shared memory is divided into *banks* that can be accessed simultaneously, yielding a high memory bandwidth as long as no bank conflict happens [34, 27].

Off-chip memory includes global memory, local memory, constant memory, and texture memory. Global memory is the largest available memory on the GPU but is also one of the slowest, together with local and texture memory. Local memory is local to each thread and is used when threads does not have any available registers left or if the data is too big to fit in registers. Texture memory is read-only memory for the device and is cached so it only needs to be read from off-chip memory once. The texture cache is optimized for two-dimensional spatial locality. Constant memory is, like texture memory, read-only cached memory that can broadcast a single value to threads in a half-warp, leading to only a single memory access if all threads requests data from the same memory location. This will generate only $\frac{1}{16}$ of the memory traffic as you would when reading from global memory [34, 27, 47].

2.2.5 CUDA and OpenCL

The CUDA C language [37] is an extension of the C/C++ languages that exposes the GPU architecture to the user through an API that allows the user to create functions that can be executed on the GPU. These functions are called kernels and are executed once per thread in parallel. The CUDA API allows the user to decide the grid and block dimensionality when calling the kernels. Inside each kernel, the user is exposed to additional built-in variables and functions that interfaces with the GPU hardware. This functionality allows the user to identify the current running thread, both local to the thread block and globally in the grid, which allows the user to divide the work between the threads. In addition, the CUDA API exposes GPU specific functionality to the user, allowing optimizations not possible through regular C/C++. The memory levels are one of these, and the user can define what kind of memory the allocated data should be placed in through special keywords like __shared__ and __global__. The API also provides method to allocate, deallocate, and transfer data between the host and the device [35].

CUDA is specific to NVIDIA GPUs and is developed by the company that also designs the hardware. This allows the CUDA API to map closely to the underlying hardware architecture and computing characteristics of the NVIDIA GPU. However, this also limits the amount of devices the code can be run on. OpenCL is an open standard for accelerated computing which provides functional portability across all devices which supports the OpenCL standard. Since OpenCL needs to support a wide range of devices, the OpenCL API is more general than the CUDA API. The programming model for CUDA and OpenCL is still very similar, and while the two APIs uses different terms, they use the same grid and thread block model for execution on the GPU. Table 2.1 gives an overview of how the indexing and dimensionality is queried in each of the APIs, and Table 2.2 shows the terminology used in CUDA and OpenCL.

Table 2.1: Difference between the CUDA and OpenCL API for index and dimension queries [25, 35].

Description	CUDA (variables)	OpenCL (functions)
Amount of blocks in the grid	gridDim	get_num_groups
Amount of threads in the block	blockDim	get_local_size
Index of the block	blockIdx	get_group_id
Local index of the thread	threadIdx	get_local_id
Global index of the thread	blockIdx * blockDim + threadIdx	get_global_id
Amount of threads in the grid	gridDim * blockDim	get_global_size

Table 2.2: Difference between the CUDA and OpenCL terminology [35, 20].

CUDA	OpenCL
Streaming multiprocessor	Compute unit
Thread	Work-item
Thread block	Work-group
Shared memory	Local memory
Local memory	Private memory
Global memory	Global memory
Constant memory	Constant memory
Texture memory	Image memory

2.3 Performance Portability Across GPUs

While the programming models used for CUDA and OpenCL provides functional portability across all their supported devices, guaranteeing the same final outcome of a computation, the performance of said computation may vary from device to device [46, 24]. An application that is optimized for a specific device might therefore perform worse on other devices, based on the underlying hardware architecture and the resources available for each device [43]. Performance portability on the GPU is often worse than on the CPU, since the GPU architecture often changes more drastically than on the CPU. In addition, many of the optimizations on the GPU are more often exposed to the user, like shared memory and block dimensions, while on the CPU they are often done in the hardware or by the compiler, like caching and prefetching. While the block distribution, as seen in Section 2.2.1, mitigates this problem across GPUs, it does not solve it.

2.3.1 Auto-Tuning

The responsibility of performance portability is thus on the user, which means it is often necessary to optimize the application for each device the application is meant to be executing on to achieve high-performance across devices. This work is not feasible for most programmers to do manually, as the amount of devices being targeted is often substantial. Additionally, the process of optimizing the application for each device itself is time consuming and prone to errors [14].

A proposed solution to the performance portability problem on the GPU is auto-tuning. Auto-tuning is the act of automatically generating implementation candidates, measuring the performance of these candidates, and then choosing the best one based on the measurements [14]. Auto-tuning has seen successful use in areas like Fast Fourier Transform (FFT) with the FFTW library [17], and linear algebra with the ATLAS library [7], among others. Auto-tuning can be done empirically, but as the amount of possible implementation candidates can be large it can be a time consuming and impractical solution.

A model-driven approach for auto-tuning is often a better solution, as it narrows down the search space before it evaluates the implementation candidates. It achieves this by introducing a performance model that applies some heuristic or domain knowledge to find a subset of the implementation candidates that are most likely to be one of the optimal solutions. However, the accuracy of model-driven auto-tuning is dependent on the quality of the model, which usually requires a lot of time and high domain-knowledge of the target architectures to implement. Machine Learning (ML) methods have been used to train the auto-tuning model instead of manually implementing it. This is done by generating implementation candidates that can be used to train an ML based performance model. This model can then be used to predict the subset of implementation candidates that is most likely to perform well, and then evaluate these [14, 28].

The ability to use ML to train a performance model, is one of the main driving factors for the development of ImageCL and its tuning parameters, which will be described in Section 2.4.

2.4 ImageCL

The OpenCL and CUDA languages exposes the user to many distinct and often complex optimization opportunities. However, while a certain combination of optimizations might lead to good performance on the device being targeted, it might lead to worse performance on another. This section describes the ImageCL language and compiler developed by Falch and Elster [12, 15]. ImageCL tries to make GPU programming easier by turning the optimization opportunities exposed in OpenCL and CUDA into tuning parameters which can be handled by an auto-tuner instead.

2.4.1 The ImageCL Language

ImageCL is a programming language designed to simplify the process of creating image processing kernels for heterogeneous hardware. The language is based on the same programming model used by OpenCL and CUDA. However, it makes some changes to the model to simplify the process. In contrast to OpenCL and CUDA, where the programmer works in a two-level thread hierarchy as seen in Subsection 2.2.1, ImageCL only defines a single flat thread space. The size and dimensionality of the thread blocks are turned into an implementation detail instead [12].

Like in OpenCL and CUDA, the ImageCL kernel represents the work performed by a single thread, but ImageCL defines the built-in variables `idx` and `idy` which represents the identity of the thread in x-, and y-direction, respectively. These built-in variables can be used to easily access the data element, also referred to as pixels, corresponding to each thread in the grid. This is in contrast to CUDA especially, where the user is exposed to the local thread identity within a thread block, in addition to the dimensionality and size of the thread blocks and the grid. This often forces the user to do the calculations manually to get the corresponding data element for a thread [12].

ImageCL also provides a custom datatype called `Image` that can be mapped to the grid, defining the size and dimensionality of the grid to be the same as the `Image`. The `Image` datatype allows for two-dimensional indexing, making it easy to apply `idx` and `idy` to access the pixels in the `Image`. In addition, the user is able to specify a boundary condition that enables out-of-bounds accesses to be well defined on `Images`. This boundary condition can either be set to clamped, meaning that the accesses outside the image is set to the closest pixel within the image, or as a constant value specified by the user.

Listing 2.1 shows an implementation of the rules of *Conway's Game of Life* [18] in ImageCL. In the example we can see how the grid is mapped to the `Image` and how the boundary condition is set to a constant value, i.e 0. It also illustrates how the built-in variables `idx` and `idy` are used to access the `Image` in a stencil pattern.

As explained in Section 2.2.4, the GPU has multiple types of memory that is exposed to the user in both CUDA and OpenCL. This makes the memory mapping complex for the user, as each memory level can affect the performance of the application in some way. ImageCL abstract away these different memory levels and exposes the user to only a single flat address space [12].

```
#pragma imcl grid(v, o)
#pragma imcl boundary_cond(v:constant)
void conway(Image<int> v, Image<int> o){
    int sum = v[idx - 1][idy - 1]
            + v[idx][idy - 1]
            + v[idx + 1][idy - 1]
            + v[idx - 1][idy]
            + v[idx + 1][idy]
            + v[idx - 1][idy + 1]
            + v[idx][idy + 1]
            + v[idx + 1][idy + 1];

    if (v[idx][idy] == 1) {
        if (sum == 2 || sum == 3) {
            o[idx][idy] = 1;
        } else {
            o[idx][idy] = 0;
        }
    } else if (sum == 3) {
        o[idx][idy] = 1;
    } else {
        o[idx][idy] = 0;
    }
}
```

Listing 2.1: An implementation of the rules of *Conway's Game of Life* [18] in ImageCL.

2.4.2 The ImageCL Compiler

The ImageCL compiler is a source-to-source compiler, or transpiler, which currently is able to generate OpenCL and CUDA code. All the complexity that the ImageCL language abstracts away, like the memory levels and the thread hierarchy, are turned into tuning parameters that each corresponds to one possible implementation in OpenCL or CUDA, where each implementation will have a distinct optimization applied.

While these tuning parameters could be given manually by the user, the ImageCL compiler is meant to be used together with an auto-tuner that is able to pick the best performing parameters for a given device. This tuning phase operates together with an auto-tuner to generate multiple implementation candidates. The source code is analyzed to find the possible parameter space, which is then fed into the auto-tuner. The auto-tuner generates the parameter values and the ImageCL compiler creates a candidate implementation that is benchmarked to find the execution time. The auto-tuner can repeat these steps multiple times until it is confident that the best implementation candidate has been found [12, 15].

2.4.3 Tuning Parameters

This section describes the tuning parameters used in ImageCL. Since ImageCL is a programming language focusing on image computation, which includes stencil based computations, the tuning parameters will be most specific towards the types of optimizations which benefits stencil computations and other computations found mostly in image processing. The following tuning parameters are currently used in the ImageCL compiler [12, 15]:

- **Thread block size:** The thread block size can play a decisive role in the performance of the application. One reason is that communication between threads often happens through shared memory, which is local to each block. So dependent on the communication pattern in the application, the optimal thread block size might vary. Another reason is that the number of blocks needed will vary based on the size of the blocks, and the amount of scheduling done by the SMs will vary based on the number of blocks used, as illustrated in Figure 2.2(b) in Section 2.2.1. The grid size however, is mapped to the data and is therefore not a tuning parameter.

- **Thread coarsening:** Thread coarsening is how much work each thread will perform. While the GPU should use thousands of threads to fully utilize the hardware, the input data might be millions of elements long. In that case it might be better to let each thread perform more work instead of spawning millions of threads. This is done by letting each thread work on a local block of pixels. The dimensionality of this local block is also part of the tuning parameter, since the access pattern of the kernel might give an performance impact based on it.

- **Thread mapping:** The thread mapping describes the access pattern used by the threads. For example, if a thread is assigned pixels in a contiguous block of memory, the access pattern could give good spatial locality. However, on the GPU this might give overall worse performance as the access pattern has poor coalescing [34, 35]. By instead using interleaved memory access it might be possible to get a performance increase as the memory access will be coalesced into fewer total memory accesses.

- **The memory parameters:** All the memory levels used as tuning parameters in ImageCL are described in Section 2.2.4 and each of them provides their own distinct benefits and drawbacks depending on the contexts they are used in. Additionally, not all memory types can be used depending on the kernel, like texture and constant memory, since they are read-only from device and can thus not be modified. By turning them into tuning parameters, ImageCL lets the auto-tuner predict if it is beneficial to use them instead of global memory in that particular case.

- **Loop unrolling:** Loop unrolling, or loop unwinding, is an optimization that repeats the loop body while adjusting the loop counter to match the total amount of work done. This can lead to reduced branch penalty from the loop, reduce the overall executed instructions, and potentially exposes more parallelism to the compiler. However, loop unrolling can also lead to increased program size which can potentially lead to an increase in instruction cache misses, hinder function inlining as the inlined code size would be multiplied with the degree of loop unrolling, and possibly lead to worse branch prediction if the iterations contained branching [33].

  Since there are both significant benefits and disadvantages of applying loop unrolling, ImageCL turns it into a tuning parameter and lets the auto-tuner decide what to do.

- **Register caching:** Register caching is a CUDA specific optimization that uses a warp-local cache built up from registers. Registers provide both higher bandwidth and lower access latency than shared memory, but using registers as a user managed cache instead of shared memory has historically not been an option due to their thread local lifetimes. However, NVIDIA's Kepler architecture introduced the shuffle instruction which made it possible for threads within a warp to share data without going through other memory levels, *e.g.* shared memory [39].

To apply register caching as an optimization, we need an access pattern which is well suited for manual caching, like stencil computations which provides a predictable access pattern at compile time. Usually, shared memory has been used for this kind of memory optimizations, but this is also a use-case well suited for register caching. Register caching also performs store-and-load operations in a single step, in contrast to shared memory, and eliminates the need for expensive thread block-wise synchronization. This provides register caching an additional performance advantage compared to shared memory as long as registers are not spilled into local memory [13].

2.4.4 Stencil Computations

Stencil computation is a kind of computation that involves determining the value of an element in a grid according to a fixed access pattern, called a stencil. Stencil computations are often found in scientific code like image processing, discretized differential equations, and adaptive mesh refinement. Stencils comes in many different variants, and Figure 2.3 shows a five-point stencil used to determine the value of a single element in a two-dimensional grid. Stencil computations are well suited for the GPU, as each element in the grid can be computed independently of each other. In addition, stencil computation uses a fixed access pattern which makes them particularly well suited for register caching since register caching needs to know the access pattern at compile time [13, 40].

The computing time and memory usage of stencil computations grows linearly with the amount of elements within an array. This means that parallel implementations of stencil computations are of high importance

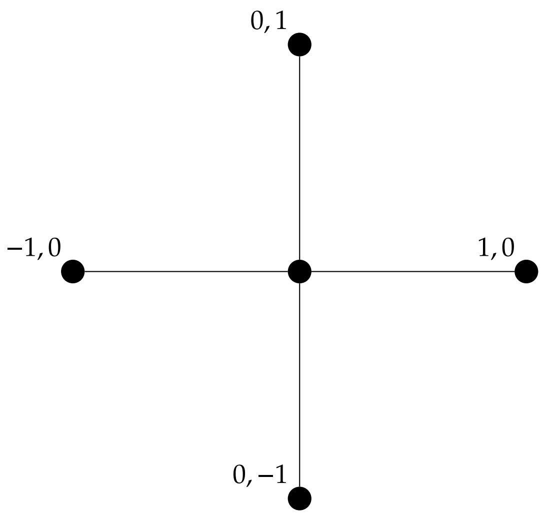


Figure 2.3: A five-point stencil.

in areas like scientific computing, where the working sets are usually very large. Stencil computations are therefore one of the main targets for the ImageCL language, and provides easy abstractions for working with these sort of computations.

2.4.5 The ROSE Compiler Infrastructure

The ImageCL compiler is implemented using ROSE [45]. ROSE is an open source compiler infrastructure to build source-to-source program transformation and analysis tools for large scale applications. It particularly targets custom tools for static analysis, program optimization, performance analysis, and cybersecurity. It consists of multiple different front-ends for different languages and currently supports C [6], C++ [16], FORTRAN [54], UPC [50], and OpenMP [41] for C, C++, and FORTRAN applications.

The intermediate representation used in ROSE is a high level Abstract Syntax Tree (AST) that is well suited for source-to-source transformations, which allows the user to do analysis, optimizations, and transformations on it. An example of a simple AST is illustrated in Figure 2.4. By modifying the AST we can insert and modify the necessary code to generate optimized GPU code which has been abstracted away and replaced by

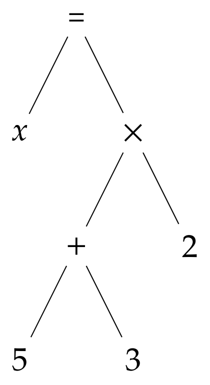


Figure 2.4: The assignment $x = (5 + 3) \times 2$ represented in an AST.

tuning parameters. When we have finished transforming the AST to represent valid OpenCL or CUDA code and applied the given optimizations, ROSE is able to traverse the AST and generate the corresponding source code. Listing 2.2 illustrates how the AST in Figure 2.4 would be built using the ROSE API.

```
auto *assignment = buildAssignStatement(
    buildVarRefExp("x", scope),
    buildMultiplyOp(
        buildAddOp(
            buildIntVal(5),
            buildIntVal(3)
        ),
        buildIntVal(2)
    )
);
```

Listing 2.2: The assignment $x = (5 + 3) \times 2$ built using the ROSE API.

The program analysis available includes call graph analysis, control flow analysis, data flow analysis, class hierarchy analysis, data- and system-dependence analysis, and MPI communication pattern analysis. The optimizations and translations functionality also includes partial redundancy elimination, constant folding, inlining, outlining, OpenMP directive lowering, automatic parallelization and loop transformations [45].

2.5 Proxy Applications

Full-scale business, scientific, and engineering applications are often very large and complex, some of them reaching millions of lines of code and having dependencies of numerous third-party libraries or frameworks. This usually leads to these applications requiring substantial systems programming expertise, domain knowledge, and time in order to being able to get the system up and running and do modification within the code base. Additionally, there are many factors that determines the performance of an application. Some of them are the hardware architecture, the runtime environment, the algorithms used in the application and their implementation, the programming language and the compiler used, and the compiler flags used when building the application. Because of this complexity it can be beneficial to use smaller applications as performance proxies when experimenting with early stage design of applications.

These smaller applications are known as proxy applications. There are multiple types of proxy applications which are useful when doing design studies of applications, each providing their own advantages and disadvantages. Some of the most common proxy applications used for performance studies are kernels, benchmarks, compact-applications, skeleton-applications, and mini-applications [10]:

- **Kernels:** Represents regions in an application which are critical for performance, *i.e. hotspots*. They are usually small, between 10 and 100 lines of code, and are usually just a single function or some other self contained region of code. While these types of proxy applications are often easy to experiment with, their smaller size limits the scope of which performance modeling can be performed on [10].

- **Benchmarks:** Proxy applications meant to provide a performance estimation for a specified software implementation. They are usually not meant for experimentation of application design and has more usage restrictions than the other proxy applications. The output of benchmarks are the runtime of the system or multiple parts of the system, which can be used to rank different implementations based on their performance or catch performance regression for a single implementation [10].

- **Compact-applications:** Simplified versions of the original application, which usually strips away the components not required to do the core logic of the application. These types of proxy applications provides good opportunities for performance modeling except for in the earliest stages of development. However, they are usually large applications in the range of $10,000$ to $100,000$ lines of code, and the ability to do experiments can in some cases be limited by the complexity of these applications [10].

- **Skeleton-applications:** Applications with an accurate implementation of the inter-process communication while the actual computations are done synthetically. This allows experimenting with communication strategies, measure the performance of the inter-process communication, and model communication complexity while keeping the code base at a manageable level [10].

- **Mini-applications:** A mini-application is a condensed implementation of one or multiple key performance impacting aspects found in the original application. This implementation is written to give the opportunity for refactoring and experimenting, while still being representative enough of the original application to be useful in the problem domain. Mini-applications are able to capture key performance issues within the original application and present it in a simplified context which can be used to do rapid testing and experimentation on. These types of proxy applications are often useful for performance modeling throughout the whole system design phase. The size of a mini-application is usually in the range of 1000 to $10,000$ lines of code [10].

2.5.1 Co-Design of Architecture and Algorithms

As clock speeds are increasingly constrained by power and cooling limits, multi-core/many-core architectures continues to dominate high performance computing [5]. The Sunway TaihuLight supercomputer in China features $10,649,600$ cores and a theoretical peak performance of $125,435.9$ TeraFLOPS [49]. However, there is a wide gap between the theoretical peak performance of these machines and what can actually be achieved with today's algorithms. As the amount of cores on the system increases, so does the complexity of the architecture and this gap is likely to grow even larger. This is one of the key design challenges for supercomputer architectures today, and one that must be overcome to reach Exascale computing (1000 TeraFLOPS).

Some of the challenges for reaching Exascale computing include the power wall [53, 52], memory wall [55], new programming models, and algorithms with good multithread-scaling. Reaching Exascale computing will require new approaches to applications, algorithms, system software, and computer architecture which faces these challenges. One of these approaches is to co-design the computer architecture and the software algorithms to create architectural-aware and highly scalable algorithms. For this to work, tools that are able to measure the impact of architectural changes on software is needed, to guide the development of future architectures and identify bottlenecks in software. This is a good use case for proxy applications, especially mini-applications, as they are able to be representative of the performance characteristics of complex full-scale applications while still being easy to modify and experiment with [1, 22, 19].

In particular, Heroux et al. (2009) [22] claims that mini-applications can benefit in the following situations:

- **Interaction with external research communities:** While the original application might be closed source and proprietary, the mini-applications are open source software. This allows easy interaction with third parties which would otherwise not have access to the software.
- **Simulations:** Mini-applications are well suited for use in simulation software as they are simpler than full-scale applications while still supporting the study of processor, memory and network architectures.
- **Early node architecture studies:** Since nodes are often available long before the complete system, mini-applications provide an opportunity to study the performance of the node architecture early in the design process of the system.
- **Network scaling studies:** Mini-applications can be configured to run on multiple processors which allows for the scalability of the network

to be studied.

- **New languages and programming models:** Since mini-applications are considerably simpler than full-scale applications, they are easily modified or rewritten into new languages and programming models.
- **Compiler tuning:** As mini-applications provides the performance characteristics of a special problem domain, it can provide compiler developers with a focused environment in which code optimizations can be developed and tested.

2.5.2 The Mantevo Project

The Mantevo project [30] was initiated in 2006 and has since then developed application performance proxies, or more specifically mini-applications, for computational science and engineering applications. Each of the mini-applications provided by the Mantevo project focuses on one or multiple key performance characteristics of an application or a class of applications within computational science and engineering. This includes partial differential equations (implicit/explicit and structured/unstructured), molecular dynamics, hydrodynamics, circuit simulations, and adaptive mesh refinement. These mini-applications provides developers of different backgrounds to have a small representative code base which can be used to understand and answer question about algorithmic implementations and architecture [10, 30].

2.5.3 Adaptive Mesh Refinement and miniAMR

Adaptive Mesh Refinement (AMR) [4, 3] is a computational method used for adapting the accuracy within certain regions of a domain. AMR is able to improve the accuracy of important sections of the domain while reducing overall memory usage in the simulation, compared to using a static domain. It is often used in science and engineering applications together with finite difference and finite volume algorithms [51].

MiniAMR is a mini-application developed by the Mantevo project in order to explore critical performance issues found in finite difference and finite

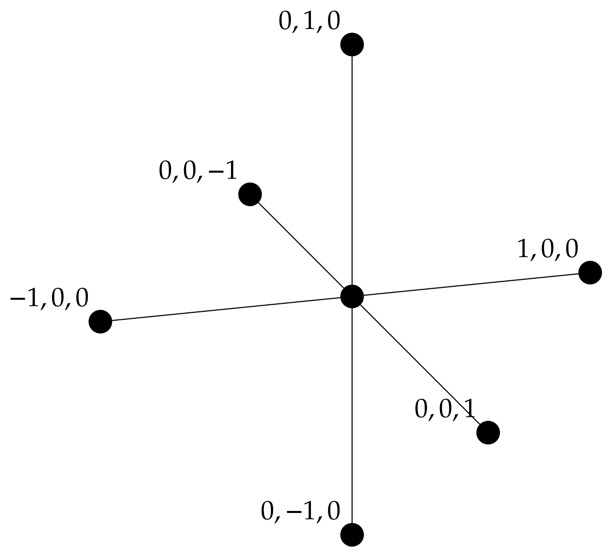


Figure 2.5: The seven-point stencil used in miniAMR.

volume applications which takes advantage of AMR. The main computation found in miniAMR is a seven point stencil computation, as seen in Figure 2.5, over the domain and yields the average value of a cell and its neighbouring cells in the x-, y-, and z-directions. While full-scale application within this domain uses multiple types of stencils based on the context, miniAMR's computation accesses all the memory associated with a block which is usually more important than the specific stencil in use. In addition, each block of the domain used by miniAMR contains the necessary ghost values needed for the computations, and needs to be communicated between all blocks with portions of a face in common for each stage of the computation. When multiple processors are used, the data is exchanged using MPI's non-blocking point-to-point functionality [51].

The miniAMR proxy application has been chosen as ImageCL is already able to handle two-dimensional stencil calculations, and extending ImageCL to handle the three-dimensional stencils found in miniAMR is a natural progression of the feature set of ImageCL.

2.6 Related Work on Performance Portability and Proxy Applications

In 2012, Kulkarni and Cavazos successfully applied machine learning to train a neural network to choose the phase ordering of compiler optimizations on a function to function basis by training on features extracted from these functions [28]. In 2016, Falch and Elster used machine learning to build a neural network performance model to predict the runtime of implementation candidates, instead of manually deriving the model [14]. This was done by creating parameterized benchmarks where each parameter represents one possible implementation of the benchmark, and train the model on the runtime of randomly chosen sets of parameters. This model could then be used to predict the runtime of new sets of parameters, instead of benchmarking each one.

Falch and Elster (2016) [12] developed ImageCL to improve performance portability by generating optimized GPU code from tuning parameters, instead of exposing them directly to the user. By combining ImageCL with an auto-tuner to pick the best implementation, a large part of the complexity of heterogeneous performance portability can be abstracted away. ImageCL was able to outperform other state of the art solution in several benchmarks.

Earlier work on the ImageCL compiler has been performed by the author as part of the specialization project in the fall of 2017 [44]. This included the implementation of a CUDA backend in addition to the already existing OpenCL backend for ImageCL, and implementing the register caching optimization for the CUDA backend.

Matthew, Simon, and Wayne [31] used TeaLeaf, a mini-application for solving the heat conduction equation, to evaluate the performance portability of emerging parallel programming models. The programming models evaluated was Kokkos, RAJA, OpenACC, and OpenMP 4.0, which was compared to the mature CUDA and OpenCL programming models. Karlin et al. (2013) [23] used LULESH, a mini-application for shock hydrodynamics, to evaluate four emerging programming models: Chapel, Charm++, Liszt, and Loci, against four established programming models: OpenMP, MPI, MPI with OpenMP, and CUDA.

Chapter 3

3D and Other Extensions to ImageCL

This chapter explains the steps of extending the ImageCL compiler with support for applying three-dimensional optimizations. The ImageCL language originally only provided the built-in `Image<T>` data type which represents a two-dimensional array of `T`s, and provides index access to data elements based on the thread identity. This is commonly applicable for stencil computation and other image manipulation functionality.

By extending the ImageCL compiler with a new built-in `Block<T>` type, representing a three-dimensional array of `T`s, three-dimensional optimizations can be applied to the data and it becomes possible to adapt three-dimensional algorithms to the ImageCL language directly.

The contributions made to the ImageCL compiler which will be discussed in the following sections are:

- The `Block` type.
- The `idz` keyword, for identifying the current thread in the z-dimension.
- The ability to have `none` as a boundary condition, which allows for manual handling of the memory accesses outside the boundary.
- Support for double precision floating point as a datatype in an `Image` or `Block`.

- Extensions to the data analysis applied by ImageCL to support three-dimensional optimizations.
- The offset operator, which allows us to alias other `Image` or `Block` arrays with an offset.
- The support of memory optimizations that can be applied to the `Block` type.

The ImageCL source-code can be found at github.com/acelster/ImageCL2.

3.1 The Block Type

While the `Image` type allows for adaptation of algorithms working on two-dimensional problem domains, it does not work for other dimensions without requiring the problem to be restructured into using a two-dimensional access pattern. The `Block` type supports three-dimensional indexing and ImageCL is able to apply the optimizations that works on individual data objects on `Block` types. This makes it possible to adapt code that uses three-dimensional access patterns without any restructuring of the code.

```
#pragma imcl grid(input)
#pragma imcl boundary_cond(input:constant)
void sum(Block<double> input, Block<double> output) {
    output[idx][idy][idz] = input[idx - 1][idy][idz]
                          + input[idx][idy - 1][idz]
                          + input[idx][idy][idz - 1]
                          + input[idx][idy][idz]
                          + input[idx][idy][idz + 1]
                          + input[idx][idy + 1][idz]
                          + input[idx + 1][idy][idz];
}
```

Listing 3.1: Using the `Block` type to write a seven-point stencil in ImageCL.

Listing 3.1 illustrates a seven-point stencil access written in the ImageCL language using the `Block` type. The boundary condition for the *input* is set to constant, which yields zero when access outside the boundary happens. For the *output* we do not need to handle writes outside the boundary as no offset is used with `idx`, `idy`, or `idz`. We also bind the grid to *input*, which is used to decide the grid size based on the size of *input* when generating the kernel wrapper. This will be described in Section 3.2.

3.1.1 The `idz` keyword

The `idx` and `idy` keywords in ImageCL provide a convenient way to get the current thread indices in x- and y-direction, respectively. This allows for access to the current element and its neighbours for the `Image` type. In addition, these keywords are used to identify thread static memory accesses in the AST and to calculate the offset used in the footprint table described in Section 3.1.4. This offset is used by ImageCL to apply the memory optimizations.

We have extended the ImageCL language with the `idz` keyword which provides the same functionality as `idx` and `idy` for the z-dimension. The thread indices in each dimension are calculated as seen in Equation 3.1, 3.2, and 3.3 when the interleaved thread mapping is used.

The $thread_{xyz}$ variables are the block local thread index, the $block_{ijk}$ variables are the block index of the current thread, and the $grid_{xyz}$ and $block_{xyz}$ variables are the size of the grid and the thread block in each respective dimension. The i, j, and k variables represents the current element being worked on by the thread and is bound by the thread coarsening given as a tuning parameter.

$$idx = thread_x + block_i \times block_x + grid_x \times block_x \times i \tag{3.1}$$

$$idy = thread_y + block_j \times block_y + grid_y \times block_y \times j \tag{3.2}$$

$$idz = thread_z + block_k \times block_z + grid_z \times block_z \times k \tag{3.3}$$

The thread indices can also be calculated using a blocking thread mapping, making the threads work on adjacent elements, using Equation 3.4, 3.5, and 3.6. The tc_x, tc_y, and tc_z variables represents the amount of thread coarsening specified by the tuning parameter, e.g. how much total work each thread will do.

$$idx = (thread_x + block_i \times block_x) \times tc_x + i \tag{3.4}$$

$$idy = (thread_y + block_j \times block_y) \times tc_y + j \tag{3.5}$$

$$idz = (thread_z + block_k \times block_z) \times tc_z + k \tag{3.6}$$

3.1.2 The none Boundary Condition

The none boundary condition is a minor extension to the ImageCL language. It allows the user to either manually handle the boundary access or even ignore it, without the compiler inserting boundary checks that could slow down the kernel. A common way to handle these kind of memory accesses is to allocate more memory than initially needed and use that memory as border values. This method uses more memory but does not require boundary checks when accessing the memory, which can be especially beneficial on the GPU as thread divergence can easily become a performance problem.

Figure 3.1 illustrates this border access where the gray blocks represents the memory allocated for the border and the yellow blocks represents the memory used for the result of the calculation. The value of the border blocks can be decided by the user and is often set to a constant value.

3.1.3 Double Precision Floating Point Support

Another improvement to the ImageCL compiler is support for the double precision floating point type. While this requires only minor additions to the ImageCL compiler for most of the kernel versions, the texture memory hardware does not currently support reading double precision floating points.

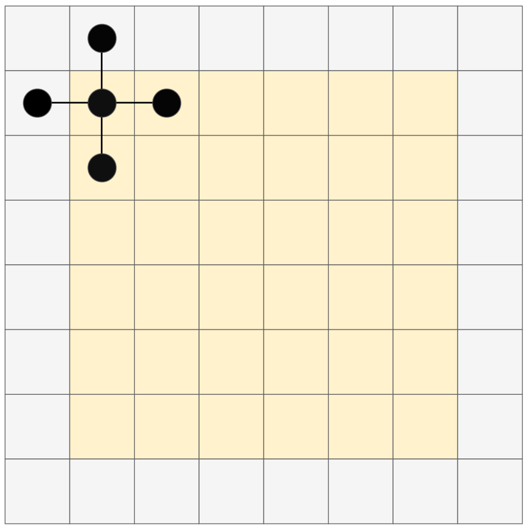


Figure 3.1: A 5-point stencil calculation accessing the border memory.

However, the CUDA backend has been extended to support texture memory with double precision floating points by generating the texture code using `int2` as the underlying datatype instead of `double`. When fetching the `int2` value from the texture in the kernel, it then casts the value to `double` using the `__hiloint2double(int,int)` function provided by CUDA. This works as long as the texture is not configured to be read using interpolation.

3.1.4 The Footprint Table

For certain optimizations to be applied by ImageCL, it is necessary to know the access pattern of the target, *e.g.* the `Image` or `Block` variable. This access pattern is used by optimizations that need to allocate additional buffers, like the shared memory optimization. In addition, it is used to decide if a boundary guard is necessary for a given access when the boundary condition for the target is set. In the ImageCL compiler this access pattern is called the footprint, and is stored in a table that maps the footprint to the variable.

```
#pragma imcl grid(input)
#pragma imcl boundary_cond(input:constant)
void sum(Block<double> input, Block<double> output) {
    output[idx][idy][idz] = 0;
    for (int z = -1; z <= 1; z += 1)
        for (int y = -1; y <= 1; y += 1)
            for (int x = -1; x <= 1; x += 1)
                output[idx][idy][idz] +=
                    input[idx + x][idy + y][idz + z];
}
```

Listing 3.2: A 27-point stencil using loops in ImageCL.

In Listing 3.1 the footprint for *input* would be $[-1, 0, 0]$ in the first access, $[0, -1, 0]$ in its second access, and so forth. The *output* would have $[0, 0, 0]$ since no offset is used when writing to it. While this example is trivial since the offsets are represented as integer literals, deciding the footprint can be challenging when the offset is represented as variables, as in Listing 3.2. By taking advantage of the liveness and constant propagation analysis features of ROSE, we can find the values of the variables used in the access as long as they are defined statically. This footprint analysis has been extended to include the expression used to access the block in the z-dimension, to support the `Block` type in addition to the `Image` type.

3.1.5 The Offset Operator

The `Block` type enables three-dimensional access in ImageCL, but higher dimensional input requires pre-processing before the kernel invocation. By extending the `Block` and `Image` types to support offsetting we can transform higher arrays input into two-dimensional `Image` arrays or three-dimensional `Block` arrays using the + operator, as seen in Listing 3.3.

```
#pragma imcl grid(input)
#pragma imcl boundary_cond(input:constant)
void sum(Block<double> data,
         int width,
         int height,
         int depth) {
    Block<double> input = data;
    Block<double> output = data + width * height *
       depth;
    output[idx][idy][idz] = 0;
    for (int z = -1; z <= 1; z += 1)
        for (int y = -1; y <= 1; y += 1)
            for (int x = -1; x <= 1; x += 1)
                output[idx][idy][idz] +=
                    input[idx + x][idy + y][idz + z];
}
```

Listing 3.3: A 27-point stencil using a single buffer to read and write using the + operator.

This is implemented by mapping all variable declaration of `Block` or `Image` arrays to the arrays used in the left hand expression together with the offset expression applied to the array. When encountering the usage of a `Image` or `Block` array later when applying optimizations, we check if this array is an alias of another array, and apply the optimization on the original array using the given offset instead of the aliased array, if necessary. This is used in optimizations like shared memory and register caching, as the buffers they use are made and initialized before the actual alias is created.

However, there are certain limitations of using this offsetting method. Certain optimizations like constant and texture memory will not work on the *input* block in Listing 3.3, since it shares the underlying buffer with *output*, which is written to. In addition, if the offset is decided by a temporary variable inside the function, *e.g.* a loop variable, the shared memory and register caching optimizations will not work. This is because when the loading of the data takes place, the variable is neither declared or initiated yet.

3.2 The Kernel Wrapper

The kernel wrapper is the function generated by ImageCL that is exposed to the user and is responsible for the GPU setup code. This includes allocating and freeing memory the input and output buffers on the GPU, transferring the input to the device and output to the host, and calling the kernel with the necessary parameters. In addition, certain tuning parameters require special initialization of the buffers used, which is also handled by the kernel wrapper, and will be described in more detail in the memory sections below.

The wrapper takes in the host buffers and their respective sizes as function parameters and uses these sizes to allocate the equivalent buffers on the device. ImageCL is able to differentiate between input buffers and output buffers by analyzing the kernel and deciding if an array is read-only, write-only, or read-and-write. If the array is read-only we only need to transfer the data from the host to the device, before calling the kernel. If the array is write-only we only need to transfer the data from the device to the host after the kernel have finished, and if it is being read and written to, we need to transfer from the host to the device before calling the kernel in addition to transfer from the device to the host after the kernel has finished.

As mentioned in Section 3.1, ImageCL can bind the grid to a `Image` or `Block` variable which is used to define the grid dimensions based on the size of the variable and the block size given as a tuning parameter. We can calculate the grid size in x-, y-, and z-dimension as seen in Equation 3.7, 3.8, and 3.9.

$$grid_x = \frac{data_x + block_x \times tc_x - 1}{block_x \times tc_x} \tag{3.7}$$

$$grid_y = \frac{data_y + block_y \times tc_y - 1}{block_y \times tc_y} \tag{3.8}$$

$$grid_z = \frac{data_z + block_z \times tc_z - 1}{block_z \times tc_z} \tag{3.9}$$

The $data_x$, $data_y$, and $data_z$ represent the size of the bound variable in x-, y-, and z-direction, respectively. $block_x$, $block_y$, and $block_z$ are the thread block sizes given as tuning parameters in each direction, and tc_x, tc_y, and tc_z are the thread coarsening for the threads in the kernel which are also given as tuning parameters. These equations find the smallest multiple of the denominator greater than or equal to the size of the `Image` or `Block` that is bound to the grid.

3.3 Global Memory

The global memory is the default memory used on the GPU, and the ImageCL code requires only slight modifications to be valid CUDA or OpenCL code. One of these modifications is to add the thread coarsening given as tuning parameter. The thread coarsening is implemented by surrounding the kernel code with one for loop for each of the dimensions, where the loop variable has an upper bound equal to the thread coarsening given as a tuning parameter. These loop variables are then included when calculating the thread indices represented by the `idx` and `idy` keywords, and the new `idz` keyword.

Another modification the ImageCL compiler performs is to transform the `Block` variables into raw pointers. This transforms the three-dimensional index access into a single one-dimensional access using the mapping seen in Equation 3.10.

$$data[x][y][z] = data[z \times data_x \times data_y + y \times data_x + x] \tag{3.10}$$

The $data_x$, $data_y$, and $data_z$ variables are the size of the *data* array in each respective dimension. In addition to adding the necessary CUDA or OpenCL specific keywords to the kernel and its function parameters, these modifications to the code are shared with most of the memory optimizations below.

3.4 Shared Memory

The shared memory optimization for the `Block` type only modifies the kernel code and requires no modifications to the kernel wrapper function. This optimization requires a separate buffer to be created, which shares its content with all threads in the same thread block. The size in each dimension of the shared memory buffer is found using Equations 3.11, 3.12, and 3.13.

$$sm_x = tc_x \times block_x + border_{left} + border_{right} \tag{3.11}$$

$$sm_y = tc_y \times block_y + border_{up} + border_{down} \tag{3.12}$$

$$sm_z = tc_z \times block_z + border_{front} + border_{back} \tag{3.13}$$

Here, tc_x, tc_y, and tc_z are the thread coarsening in each dimension given as tuning parameters, and the $block_x$, $block_y$, and $block_z$ variables are the thread block size in each dimension which are also given as tuning parameters. The *border* variables used in the equations defines how large the border size, or halo size, of the shared memory buffer should be. This is found using the footprint of the optimization target, as explained in Section 3.1.4, where the biggest offset in each dimension is chosen to be used as the border size of the buffer to assure that the memory accesses never goes outside the boundary.

After declaring the shared memory buffer we load the data from the optimization target into the buffer. A loading loop above the rest of the kernel code maps the local thread index to the global thread index, and loads the global data into the shared memory buffer. The mapping from the local to the global thread index is done as seen in Equations 3.14, 3.15, and 3.16.

$$global_x = block_i \times block_x + local_x - border_{left} \tag{3.14}$$

$$global_y = block_j \times block_y + local_y - border_{up} \tag{3.15}$$

$$global_z = block_k \times block_z + local_z - border_{front} \tag{3.16}$$

The $block_i$, $block_j$, and $block_k$ are the current thread block index in x-, y-, and z-dimension, respectively. By multiplying the current thread block index with the block size in the corresponding dimension, we get the global thread index of the first thread in the current thread block, which can then be added to the local thread index. In addition, the border size in the negative direction needs to be subtracted to handle the border correctly when reading from global memory.

After the shared memory buffer has been initialized, we replace all usages of the original global memory buffer with the new shared memory buffer. In addition to changing the access buffer, we also change the indexing used to access the buffer from global scope to thread block scope. When interleaved memory access is used we find the local indices using Equations 3.17, 3.18, and 3.19.

$$local_x = thread_x + block_x \times i + border_{left} \tag{3.17}$$

$$local_y = thread_y + block_y \times j + border_{up} \tag{3.18}$$

$$local_z = thread_z + block_z \times k + border_{front} \tag{3.19}$$

Here, the *thread* variables are the thread block local index of the current thread in each dimension, and i, j, and k represents the loop variables in the thread coarsening loops surrounding the kernel code, as mentioned in Section 3.3. By jumping a whole block length for each data element we get an interleaved memory access for the threads.

Blocking memory access is also supported, and the thread calculation instead becomes as seen in Equations 3.20, 3.21, and 3.22. By only moving a single data element per iteration, the thread will instead work on data elements adjacent to each other.

$$local_x = thread_x \times tc_x + i + border_{left} \tag{3.20}$$

$$local_y = thread_y \times tc_y + j + border_{up} \tag{3.21}$$

$$local_z = thread_z \times tc_z + k + border_{front} \tag{3.22}$$

By querying the ROSE AST for all memory accesses in the kernel we replace all occurrences of the original buffer with the shared memory buffer and replace the original indices with the thread block local indices.

3.5 Texture Memory

Both CUDA's texture memory and OpenCL's image memory provide their own memory access API that differs from the standard indexing used in the other memory levels. Three-dimensional texture memory in CUDA is accessed using the `tex3D<T>` function and OpenCL provides the `read_image` functions for supported types. These functions allow three-dimensional indexing directly without having to transform them into a single one-dimensional access, and by traversing the AST looking for memory accesses of the original array, we can replace them with the provided texture functions.

In the kernel wrapper we generate the code for binding the texture object to the optimization target as illustrated in Figure 3.2. This setup code for CUDA involves transferring the data to a three-dimensional `cudaArray_t` data type, initializing and setting up a resource description and a texture description for the texture object, before finally binding the texture object to the data in the `cudaArray_t` data type. The wrapper code for OpenCL works mostly the same as the CUDA implementation, and we initialize a three-dimensional `image_t` type by declaring it as an `CL_MEM_OBJECT_IMAGE3D` through the OpenCL API.

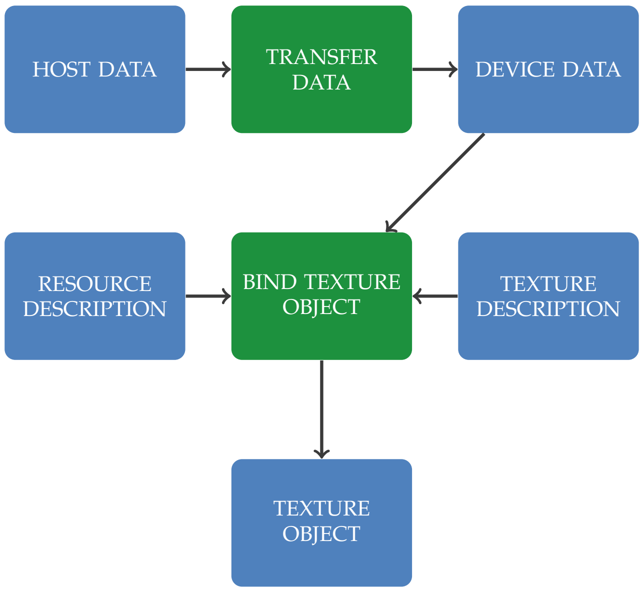


Figure 3.2: Workflow for binding host data to a three-dimensional texture object in CUDA.

When binding the texture object we can decide how it should behave when memory accesses happens outside the boundary, which allows us to potentially avoid boundary tests inside the kernel code.

3.6 Constant Memory

Both in OpenCL and CUDA, constant memory requires only minor changes compared to the global memory version. In OpenCL we mark the optimization target with a `__constant` modifier inside the kernel, and in the generated wrapper we initiate the device memory as read-only. The CUDA implementation requires the constant memory to be declared using the `__constant__` modifier in global scope, instead of passing it as an input parameter, while the kernel wrapper transfers the data to the constant memory. As with the global memory, all three-dimensional accesses are transformed into one-dimensional accesses, using the mapping from Equation 3.10.

In OpenCL the constant memory size is declared at runtime, which means that we can decide the size based on the input from the caller. In CUDA however, the constant memory size needs to be known at compile time, and can thus not be calculated from the input at runtime. This is handled by initializing the constant memory to the grid size if the user has specified the size manually using the `grid_size` attribute. Otherwise the constant memory is initialized to zero, which yields a compilation error when building the code since zero-sized variables are not allowed in device code, and forces the user to manually set the size.

3.7 Register Caching

As mentioned in Section 2.4.3, register caching is a CUDA specific optimization that works similarly to the shared memory optimization, as it requires a separate buffer. The buffer is only accessible between threads in the same warp, and the buffer size needs to be the same size as the warp. For the `Image` implementation of register caching, this allowed for a cache size of 8×4, which would have enough space for the border as well as the calculated values.

The register caching optimization was added to the `Image` type for the CUDA backend as part the authors specialization project in the fall of 2017 [44]. However, for the `Block` implementation of register caching, the warp size is too small as a cache size of $4 \times 4 \times 2$ does not leave space for the calculated value in z-direction. This means that the register caching optimization is currently not supported for the `Block` type.

Chapter 4

Applying Code Generated by ImageCL into miniAMR

This chapter describes the integration of the GPU code generated from ImageCL kernels into miniAMR. The ImageCL kernels are implementations of the seven-point stencil illustrated in Figure 2.5. Section 4.1 describes the integration of memory management and setup of the GPU code generated as part of the kernel wrapper from ImageCL. Section 4.2 explains the ImageCL implementations of the stencil code and Section 4.3 presents the kernels generated by ImageCL using tuning parameters. The miniAMR source-code can be found at github.com/acelster/miniAMR-NTNU.

4.1 Memory Management and Setup

Listing 4.1 gives an overview of the computations in miniAMR where we have integrated code generated by ImageCL. To avoid unnecessary memory allocations and memory transfers from host to device, the device memory is initialized outside the innermost loop which calls the kernel code. This is done directly after the ghost values has been communicated between the blocks to assure that the device data is synchronized for each stage in the calculation.

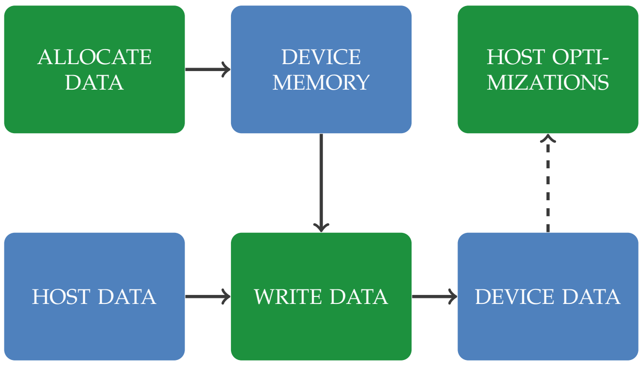


Figure 4.1: General flow of device initialization in miniAMR.

```
for timestep in timesteps do
    for stage in stages do
        communicate ghost values between blocks
        gpu init
        for var in variables do
            call stencil kernel with var
        end for
        gpu free
    end for
end for
```

Listing 4.1: Overview of miniAMR after integration of generated code.

The `gpu init` and `gpu free` methods are different depending the kernel used in the stencil calculation. However, the general flow includes allocating the device memory and fill it with the data needed in the innermost stencil loop. Host optimization can also be applied at this stage if required by the kernel implementation. This general setup code is illustrated in Figure 4.1. In the `gpu free` method, the data is transferred back to the host and the resources are cleaned up.

The mesh stored in miniAMR is divided into blocks where each block has a number of variables which again stores the three-dimensional data. This effectively makes the domain of miniAMR a five-dimensional array where the stencil calculation is applied multiple times on the three innermost levels of the array. MiniAMR stores the data using non-contiguous memory between each dimension which is not well suited for the GPU kernels which assumes the memory is stored contiguous in memory. Therefore, when the memory is written from host to device it is mapped from non-contiguous to contiguous memory, and mapped back when writing back the memory to the host.

4.2 Kernel Implementations

Listing 4.2 shows pseudo code of the reference implementation of the stencil calculation found in miniAMR. The seven-point stencil is applied to each value in the block and the result of the calculation is written to a temporary buffer to avoid reading newly updated value. The values are then written back to the original buffer in a second pass across the block. The reference implementation avoids boundary checks by allocating additional memory for the x-, y-, and z-dimensions that is used for the border.

When the stencil calculation is performed on the GPU, writing to a temporary buffer as seen in the reference implementation is not enough to avoid reading newly updated values. This is because some threads might write back the values in the second pass before other threads have read the values in the first pass. While the GPU provides instructions to synchronize the threads, this only works within each thread block. This thread synchronization might also reduce the performance of the kernel as it might stall threads.

We have created three versions of the reference implementation that solves this synchronization problem in different ways, each of them providing their own benefits and drawbacks. These versions will be described in the following sections.

```
for block in blocks do
    for x in 1...block_size_x do
        for y in 1...block_size_y do
            for z in 1...block_size_z do
                tmp[x][y][z]
                    = block[var][x - 1][y][z]
                    + block[var][x][y - 1][z]
                    + block[var][x][y][z - 1]
                    + block[var][x][y][z]
                    + block[var][x][y][z + 1]
                    + block[var][x][y + 1][z]
                    + block[var][x + 1][y][z]
                tmp[x][y][z] /= 7.0
            end for
        end for
    end for
    for x in 1...block_size_x do
        for y in 1...block_size_y do
            for z in 1...block_size_z do
                block[var][x][y][z] = tmp[x][y][z]
            end for
        end for
    end for
end for
```

Listing 4.2: Pseudo code of the reference implementation of the kernel.

4.2.1 Using Single Read-Write Buffer

The kernel only works on one variable each time it is called. The first kernel implementation takes advantage of this by allocating space for one additional variable, which can be written to using an offset different from the one being read from. Listing 4.3 shows how this offset is used in the ImageCL kernel using the offset operator.

```
int r_offset = (block * (num_vars + 1) + (var + 1))
             * width * height * depth;
int w_offset = (block * (num_vars + 1) + var)
             * width * height * depth;
Block<double> read  = input + r_offset;
Block<double> write = input + w_offset;
write[idz + 1][idy + 1][idx + 1]
    = (read[idz][idy + 1][idx + 1]
    +  read[idz + 1][idy][idx + 1]
    +  read[idz + 1][idy + 1][idx]
    +  read[idz + 1][idy + 1][idx + 1]
    +  read[idz + 1][idy + 1][idx + 2]
    +  read[idz + 1][idy + 2][idx + 1]
    +  read[idz + 2][idy + 1][idx + 1]) / 7.0;
```

Listing 4.3: Simplified ImageCL kernel using a single read-write buffer

The write buffer is created using an offset on the input buffer that finds the position of the variable data right before the data being read. This means that the data being written and the data being read never overlaps and the threads does not need to synchronize. Like the reference implementation, we allocate extra memory for the border to avoid boundary checks. This means we need to shift the memory accesses on the `Block` types by one to start within the borders. The full implementation of the ImageCL kernel can be found in Appendix A.

While this implementation is the most memory efficient version of the ImageCL kernels, the optimizations that can be applied to it is limited. This is because the aliased array is being read and written to, which prohibits optimizations that requires constant data. In addition, the offset is calculated using values that are modified inside the kernel (the `block` variable in Listing 4.3 is a loop counter), which makes shared memory unable to load the data needed into its buffer.

4.2.2 Using Read and Write Buffers

The second kernel implementation uses two separate underlying buffers, one for the input data that is being read and one for the output data. The offset operator is used to transform the original five-dimensional arrays into three-dimensional `Block` arrays. Listing 4.4 shows a simplified version of this kernel and the full implementation can be found in Appendix A.

```
int offset = (block * num_vars + var)
           * width * height * depth;
Block<double> read = input + offset;
Block<double> write = output + offset;
write[idz + 1][idy + 1][idx + 1]
    = (read[idz][idy + 1][idx + 1]
    +  read[idz + 1][idy][idx + 1]
    +  read[idz + 1][idy + 1][idx]
    +  read[idz + 1][idy + 1][idx + 1]
    +  read[idz + 1][idy + 1][idx + 2]
    +  read[idz + 1][idy + 2][idx + 1]
    +  read[idz + 2][idy + 1][idx + 1]) / 7.0;
```

Listing 4.4: Simplified ImageCL kernel using a read and write buffer.

By using separate underlying buffers for the read and write arrays, ImageCL is able to apply the constant memory optimization to the data being read. However, shared memory can not be applied to this version either, since the offset is dependent on the current iteration in the loop, *i.e.* the current block.

4.2.3 Using 3D Read and Write Buffers

For ImageCL to be able to apply all memory optimizations, the kernel need to use separate read and write buffers and the buffers need to be three-dimensional. The last kernel implementation only contains the stencil code for a single block and variable. This requires the array parameters to be pre-processed into three-dimensional arrays before calling the kernel, and the kernel needs to be executed multiple times using different offsets. The code for the last kernel can be seen in Listing 4.5.

```
output[idz + 1][idy + 1][idx + 1]
    = (input[idz][idy + 1][idx + 1]
    +  input[idz + 1][idy][idx + 1]
    +  input[idz + 1][idy + 1][idx]
    +  input[idz + 1][idy + 1][idx + 1]
    +  input[idz + 1][idy + 1][idx + 2]
    +  input[idz + 1][idy + 2][idx + 1]
    +  input[idz + 2][idy + 1][idx + 1]) / 7.0;
```

Listing 4.5: ImageCL kernel using three-dimensional read and write buffers.

Some of the memory optimizations require considerable setup code on the host side. Since the offset used in these kernels are decided by the current block and variable being worked, we need to reapply these optimizations each time one of those changes. This can lead to notable overhead when using optimizations like constant memory or texture memory, as their setup code contains memory transfers. The full implementation can be found in Appendix A.

4.3 Generated Kernels

Using the ImageCL kernels in Section 4.2, we have generated multiple GPU implementations from different tuning parameters. The number of potential implementations that can be generated from the tuning parameters are too large to be reasonably able to integrate all of them into miniAMR. We have focused on the memory optimizations and generated six CUDA kernels using different access methods and memory optimizations. The generated implementations can be seen in Table 4.1.

Three global memory versions have been generated (v1, v2, and v3), one for each of the ImageCL kernels. In addition, the three-dimensional ImageCL kernel have been used to generate a shared memory (v4), a constant memory (v5), and a texture memory (v6) implementation.

Table 4.1: The configurations and tuning parameters used for each generated kernel.

Configuration	v1	v2	v3	v4	v5	v6
ImageCL Kernel	v1	v2	v3	v3	v3	v3
Thread Coarsening	1	1	1	1	1	1
Block Size	8x8x8	8x8x8	8x8x8	8x8x8	8x8x8	8x8x8
Interleaved	True	True	True	True	True	True
Shared Memory				input		
Constant Memory					input	
Texture Memory						input
Language	CUDA	CUDA	CUDA	CUDA	CUDA	CUDA

The kernel wrappers generated by ImageCL can not be used directly, as the setup required for the GPU has been moved outside the innermost variable loop, as seen in Listing 4.1. This has been done to lower the setup overhead by reusing memory and having fewer larger memory transfers between the host and the device. Instead, we use the generated wrapper as a baseline and manually adapt the setup code to fit with the miniAMR code structure.

Chapter 5

Results and Discussion

In this chapter we present and discuss the performance measurements of the kernels generated in Chapter 4. Section 5.1 describes the methodology used to get the performance measurements. Section 5.2 presents the performance measurements and Section 5.3 discusses the results.

5.1 Methodology

Table 5.1 shows the specifications of the three systems used to benchmark the miniAMR versions. The miniAMR versions have been run single threaded on the CPU, together with a single GPU. Since we generate CUDA implementations of the kernels, only NVIDIA GPUs have been used in the benchmarks.

The GPU kernels have been measured using the events provided by the CUDA API, and the setup code has been timed using the `gettimeofday` function provided by `time.h`. The performance measurements are the arithmetic average of ten runs. The naming of the different implementations in the following sections references the kernels in Table 4.1.

Table 5.1: The specifications of the three systems used in the measurements.

Hardware	System 1	System 2	System 3
CPU	Intel® Core™ i7-6700K 4.00 GHz	Intel® Xeon™ E5-2695 v4 2.10GHz	Intel® Core™ i7-7700K 4.20 GHz
Memory	16 GB	128 GB	32 GB
GPU	NVIDIA GTX 980	NVIDIA Tesla P100	NVIDIA Titan V
OS	Ubuntu 16.04 64-bit	CentOS Linux release 7.4.1708	Ubuntu 16.04 64-bit

Table 5.2: The default problem size in miniAMR.

Variable	Value
Max number of blocks	500
Number of variables	40
X block size	10
Y block size	10
Z block size	10

5.1.1 Configurations

When measuring the performance of the kernel implementations, we use the default configurations of miniAMR and only modify the problem size in x-, y-, and z-dimension. Table 5.2 shows the default problem size used in miniAMR. As we increase the block size in x-, y-, and z-dimension, the memory usage grows fast. The largest problem size we could calculate on all systems was when x, y, and z was 40. All the configurations values of miniAMR used to gather the performance measurements are given in Appendix B.

5.1.2 Validation

To assure that the generated kernels and their optimizations have been generated correctly, we perform a validation run before running the benchmarks. This validation run dumps the whole mesh structure in miniAMR to file after each time step in the calculation. This is done for all the kernel versions, including the reference version. Afterwards, the mesh dump of the generated kernels are compared to the mesh dump of the reference version, to assure that the same results are calculated at each time step.

5.2 Performance Measurements

This section presents the performance measurements of the miniAMR application using the generated kernels. The generated kernels are compared to the reference version written in the `C` language, and to each other. The *Dimension* axis in the graphs presented below, represents the value of the x-, y-, and z-block size in miniAMR. The *Time* axis represents the time in seconds. All kernels have been tested using a uniform block dimension from 4 up to 40 (skipping odd numbers), except for constant memory which can only fit sizes of $18 \times 18 \times 18$ and below in its memory. The detailed performance measurements can be found in Appendix C.

5.2.1 Execution Time

Figure 5.1 illustrates the total time spent inside the GPU kernel for the three systems. The texture memory version (v6) has the best performance on all the systems tested on, and shared memory (v4) starts to outperform the other kernels on larger dimensions, except on the Titan V system. The remaining kernel implementations performs about the same, with the exception of constant memory (v5) which performs considerably worse on all systems. The Titan V system has the best overall performance, with the P100 system trailing close behind.

Figure 5.2 shows the amount of time spent calling the kernel from the C code. While Figure 5.1 only includes the time spent inside the kernel code, Figure 5.2 also includes the local optimizations and preprocessing that is required before actually calling the kernels. For the GTX 980 and Titan V systems, the GPU kernels outperforms the reference version when the dimension is around $16 \times 16 \times 16$. For the P100 system, the GPU kernels outperforms the reference version between 20×20×20 and 24×24×24. In the GTX 980 and Titan V system, the time used by the reference implementation grows fast up to 30×30×30, before dropping significantly at 32×32×32. The reference implementation on the P100 system has a more steady growth in time usage, without a drop at $32 \times 32 \times 32$.

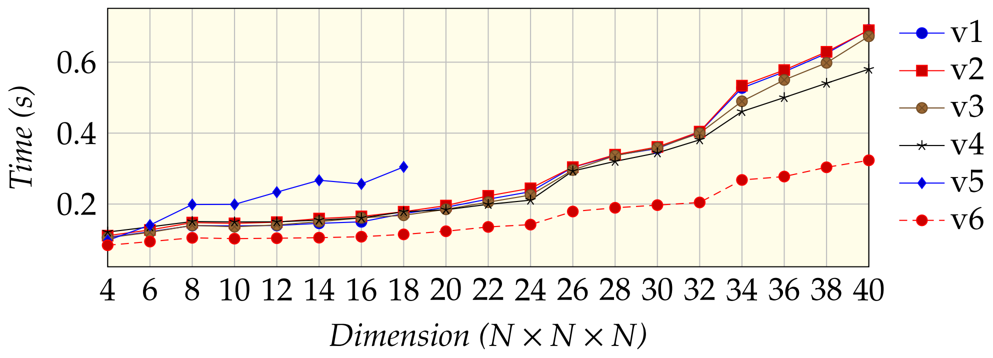
Figure 5.3 illustrates the total GPU setup time required by each of the kernel implementations. As seen from the figure, the time spent in the setup code is substantial compared to the time spent in the actual kernel. As expected, the single read-write buffer implementation (v1) has a slightly lower setup time since it allocates fewer buffers than the other implementations. The texture memory version (v6) has the highest setup time of the kernels.

Figure 5.4 shows the total runtime of the miniAMR implementations. While the kernel execution is improved on the GPU versions when the problem size reaches a certain dimension, the setup time required by the GPU is currently too large to see a performance improvement in the whole miniAMR application. The single read-write buffer implementation (v1) performs best out of the GPU implementations, as the timing is dominated by the setup time.

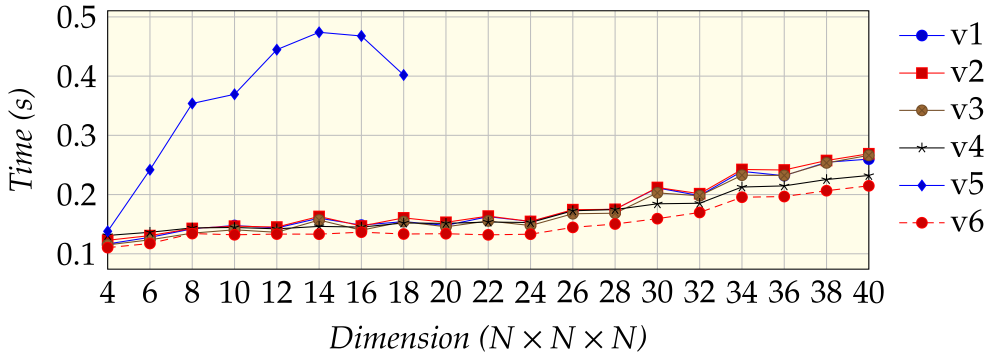
5.2.2 Memory Usage

Figure 5.5 and Figure 5.6 illustrates the amount of memory allocated and the amount of memory copied between the host and the device. The single read-write buffer (v1) implementation allocates and transfers significantly less memory than the other implementation. The amount of memory allocated is nearly half of the other implementations, and the amount copied is roughly 30% less than the other versions.

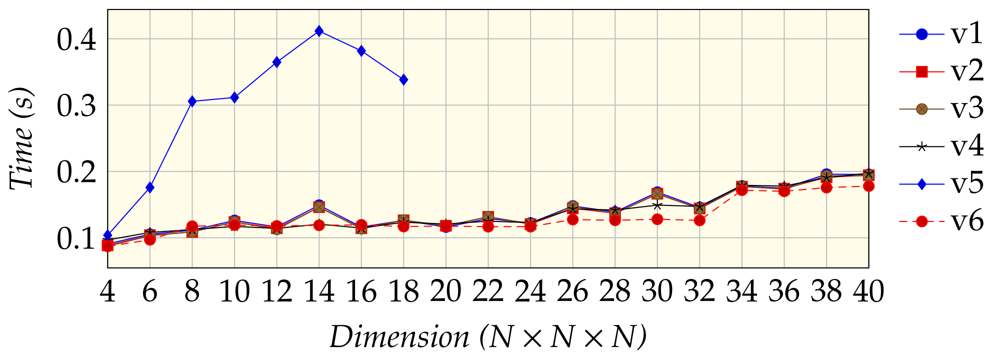
This reduces the amount of setup time required on the GPU for this implementation, which is the main contributor to this implementation of miniAMR being the overall fastest of the GPU implementations. The remaining miniAMR implementations allocates and copies roughly the same amount.



(a) System 1

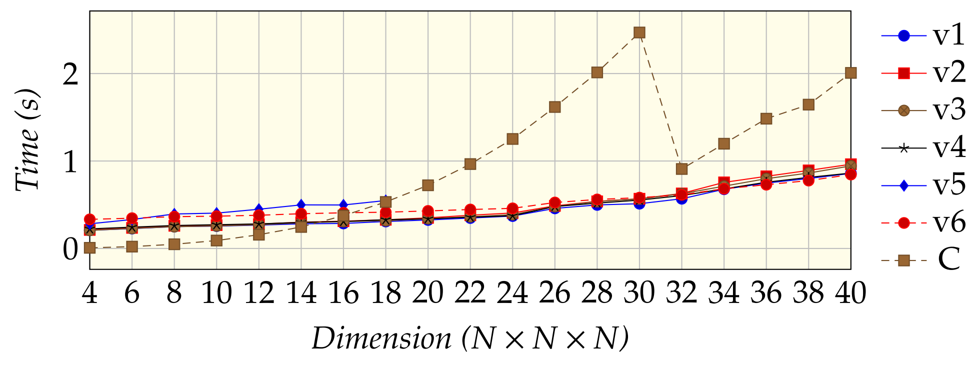


(b) System 2

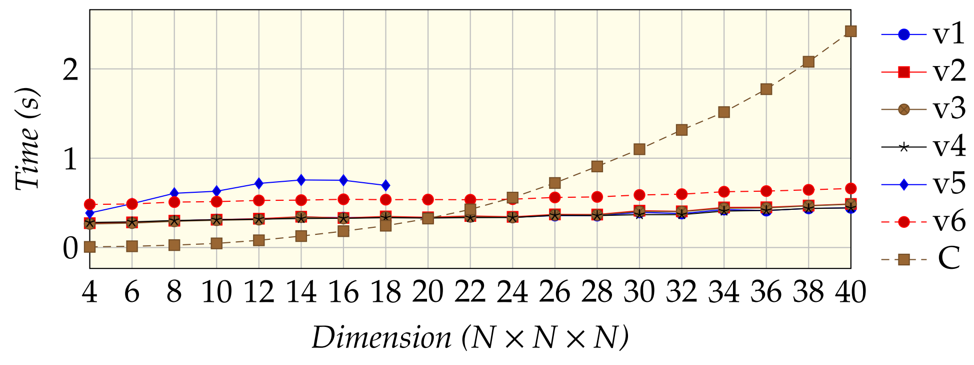


(c) System 3

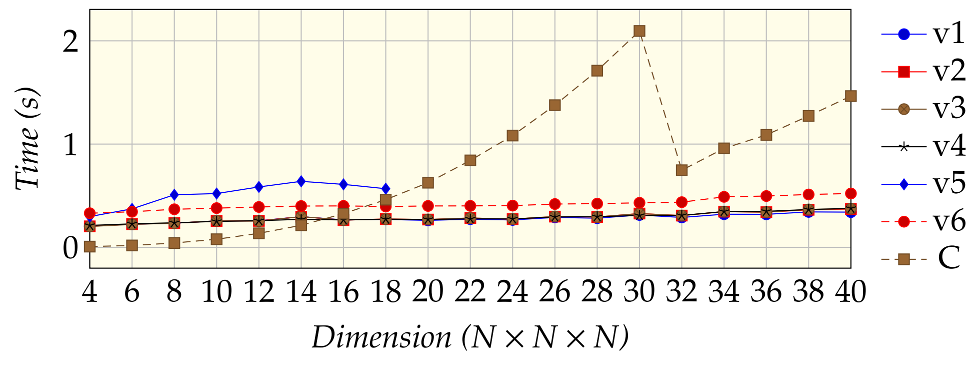
Figure 5.1: Time spent inside the kernel code for the GTX 980-, P100-, and Titan V-system. Timed using CUDA events.



(a) System 1

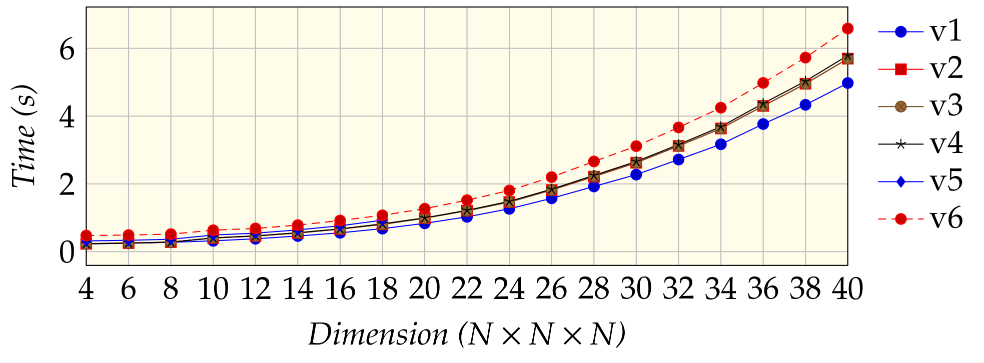


(b) System 2

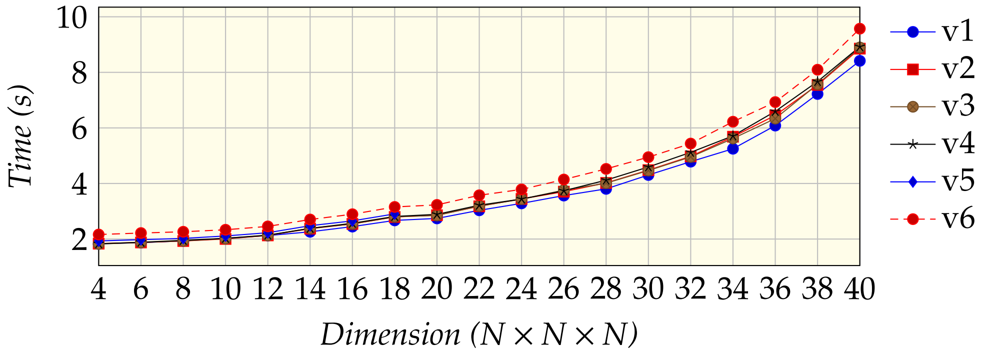


(c) System 3

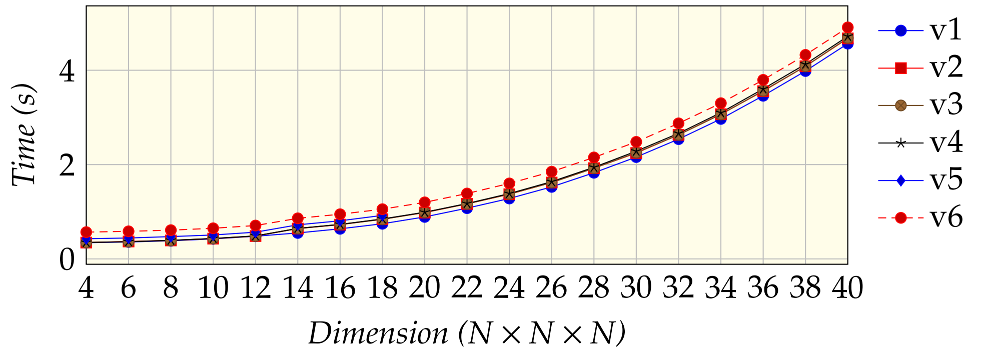
Figure 5.2: Kernel execution time including local kernel setup for the GTX 980-, P100-, and Titan V-system. Timed using `gettimeofday`.



(a) System 1

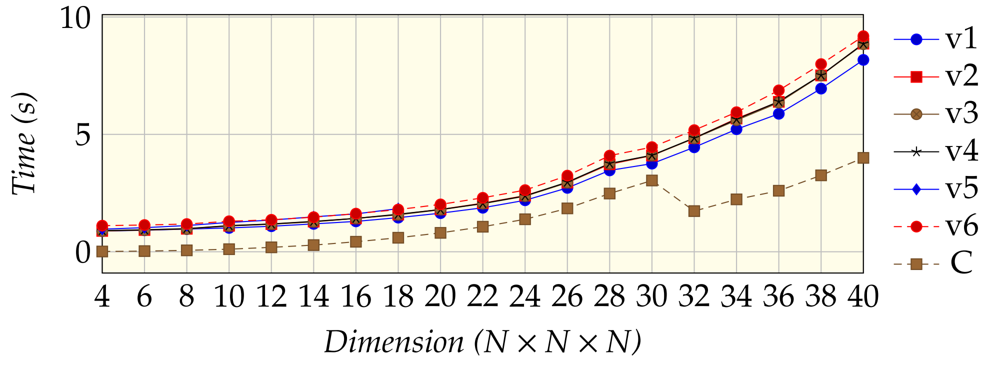


(b) System 2

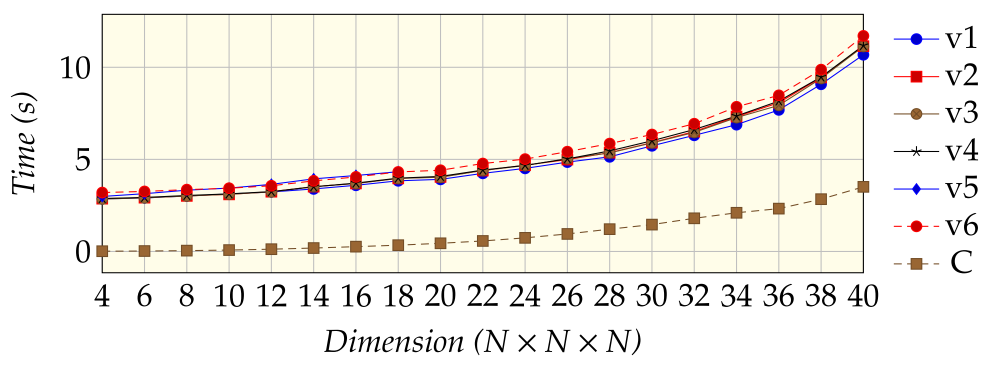


(c) System 3

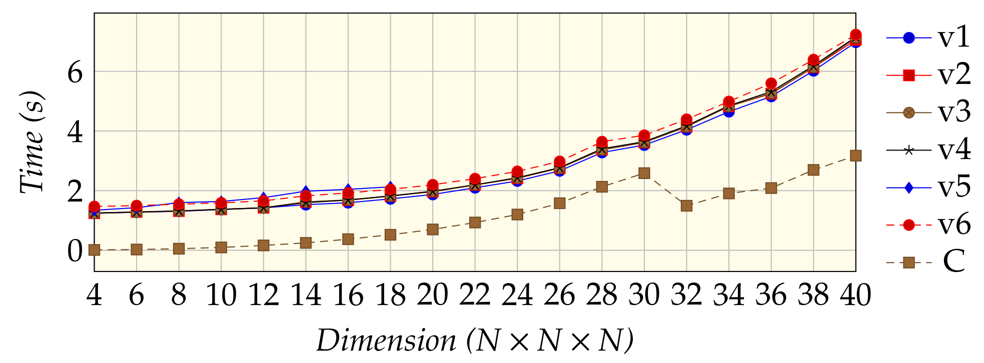
Figure 5.3: GPU total setup time on the GTX 980-, P100-, and Titan V-system. Timed using `gettimeofday`.



(a) System 1



(b) System 2



(c) System 3

Figure 5.4: Total time of miniAMR for the GTX 980-, P100-, and Titan V-system.

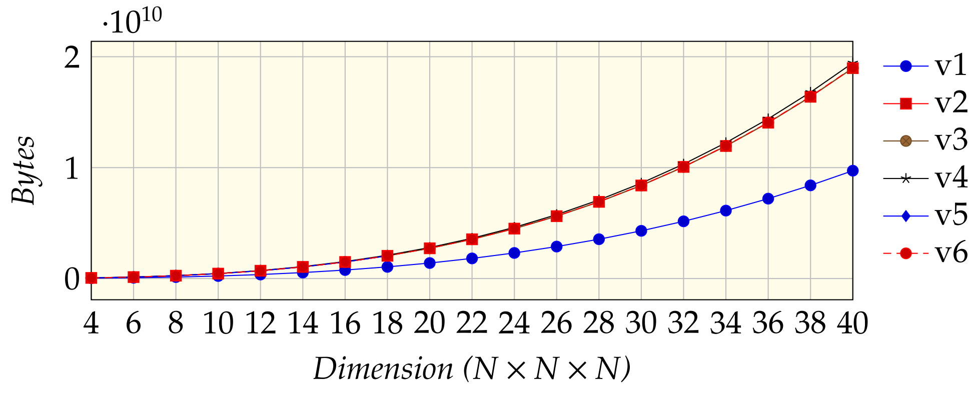


Figure 5.5: Amount of global memory allocated on the GPU.

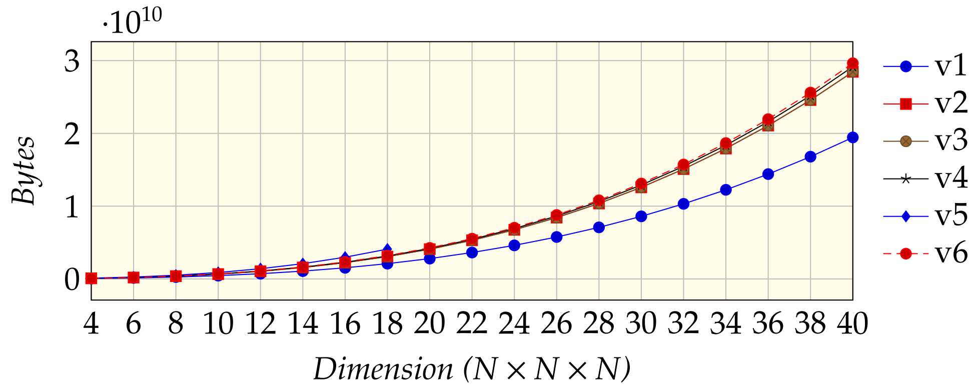


Figure 5.6: Amount of memory copied from host to device and from device to host.

5.3 Discussion

The previous sections presented the performance measurements of the various implementations of miniAMR. This section discusses the results and the advantages and disadvantages of the different miniAMR implementations.

As illustrated by Figure 5.4, none of the GPU implementations were able to speed up the whole applications, as the overhead of the memory allocations and transfers are much larger than the benefits we get from a faster kernel. The overhead of the setup code can be reduced by moving the code further up in the miniAMR application. This would, however, require a larger part of miniAMR to be able to run on the GPU, including the ghost value communication between blocks, which is currently being performed using MPI. ImageCL is currently not able to handle such general cases and this have been outside the scope of this thesis as the goal is to investigate how ImageCL is able to integrate with and improve the performance of miniAMR, not to gain the best performance increase possible.

From Figure 5.3 we can see that the difference in setup time between the systems are large, with the P100 system having a notable larger setup time than the two other systems, even though the P100 GPU is significantly more powerful than the the GTX 980 GPU. These results mirrors the single thread performance of the CPU in the systems, and it is likely that this is one of the main contributors to this performance gap between the systems.

As Figure 5.1 show, the texture memory kernel has the best performance when not including the overhead of binding the texture object, which is done as part of the local setup before calling the kernel. This is to be expected, as the access pattern performed by the seven-point stencil is well suited for texture memory. The shared memory kernel is the second best performing implementation, especially at the larger dimensions for the GTX 980 and P100 systems. The difference between the remaining implementations and the shared memory implementation is much smaller on the Titan V system. This could be the result of the enhanced L1 Data Cache introduced in Volta [29], which narrows the gap between kernels using shared memory and automatic cache.

When comparing the kernel execution time found with `gettimeofday` we can see that all the GPU kernels, except for the constant memory kernel, outperforms the reference implementation when the problem size grows large enough. This happens when the dimension is between $16 \times 16 \times 16$ and $24 \times 24 \times 24$, depending on the system. At $30 \times 30 \times 30$, System 1 and System 3 gains a speedup of 4.46 and 6.78 for the shared memory versions compared to the reference implementation, respectively. At $40 \times 40 \times 40$, System 2 is able to achieve a speedup of 5.45 for the shared memory version, compared to the reference implementation.

Additionally, the tuning parameters used to generate the GPU kernels have been manually selected and have a limited range. To achieve even better performance, an auto-tuner could be applied to pick a better performing combination of tuning parameters.

Chapter 6

Conclusion and Future Work

This final chapter first concludes the work presented in this thesis and highlights some of the results, before we will discuss and suggest possible areas suitable for future work.

6.1 Conclusion

In this thesis we extended the ImageCL language and compiler to support a broader range of applications. The extensions have been guided by the miniAMR application, an Adaptive Mesh Refinement proxy application with a seven-point stencil kernel. The extensions have focused mainly on three-dimensional optimizations through the `Block` data type.

Three kernel implementations representing the miniAMR stencil kernel were written in the ImageCL language: the single read-write buffer version, the read and write buffer version, and the three-dimensional read and write buffer version. These ImageCL kernels have all been written to take advantage of the new features implemented in the ImageCL compiler. From these ImageCL kernels, six GPU implementations were generated from different tuning parameters, and integrated into the miniAMR application. The kernels generated from ImageCL have focused mainly on the different memory optimizations and the distinct methods to handle the data buffers.

The GPU setup proved to be too large for a performance improvement of the whole application. With a more extensive refactoring of miniAMR it should be possible to reduce this overhead significantly. However, as ImageCL focuses mainly on stencil based image processing applications, using ImageCL further in the miniAMR application would be problematic without further improvements to the compiler. Some improvements that could help make ImageCL work in more general cases are given in Section 6.2.

The benchmarks shows that most of the generated GPU kernels achieves a significant performance improvement compared to the reference implementation as the problem size grows larger. At most, the shared memory implementation achieves a speedup of 4.46, 5.45, and 6.78 for System 1, System 2, and System 3 compared to the reference implementation, respectively.

6.2 Future Work

The work performed in this thesis presents several interesting areas for future work:

- The ImageCL language and compiler can be further extended with enhanced data analysis and tuning parameters to support an even broader range of applications. This might include supporting the shared memory optimization on aliased types, or extending ImageCL's register caching optimization capabilities to work with three dimensional data.
- The tensor cores and independent thread scheduling introduced in NVIDIA's Volta architecture might open up for a whole new set of optimizations, which can potentially be turned into tuning parameters in ImageCL.
- Making the ImageCL memory optimizations work on any type of input data, not just two- and three-dimensional data, would allow more general algorithms to be adapted to ImageCL. The offset operator allows this to an extent, but comes with certain limitations.

- Proper synchronization primitives is lacking in the ImageCL language, and while the compiler is able to automatically insert synchronization calls in certain cases, *e.g.* for the shared memory optimization, having explicit synchronization primitives would be beneficial for more general applications, where the compiler is not always able to figure out when synchronization is needed.
- Evaluating ImageCL using other proxy applications might give insight into further areas that can be improved. A noteworthy proxy application is TeaLeaf [8], a linear heat conduction solver that uses a five-point stencil in two-dimensions (three-dimensional support in beta), that has earlier been used to evaluate other parallel programming models [31].

Bibliography

[1] Ken Alvin et al. "On the Path to Exascale". In: 1 (Jan. 2010), pp. 1–22.

[2] AMD. *It's HIP to be Open*. URL: `https://www.amd.com/Documents/HIP-Datasheet.pdf` (visited on 04/23/2018).

[3] Marsha J Berger and Joseph Oliger. "Adaptive mesh refinement for hyperbolic partial differential equations". In: *Journal of Computational Physics* 53.3 (1984), pp. 484–512. ISSN: 0021-9991.

[4] M.J. Berger and P. Colella. "Local adaptive mesh refinement for shock hydrodynamics". In: *Journal of Computational Physics* 82.1 (1989), pp. 64–84. ISSN: 0021-9991.

[5] Shekhar Y. Borkar et al. "Platform 2015: Intel Processor and Platform Evolution for the Next Decade". In: (2005).

[6] *C - Approved standards*. URL: `http://www.open-std.org/JTC1/SC22/WG14/www/standards` (visited on 07/24/2018).

[7] R Clint Whaley and Antoine Petitet. "Minimizing development and maintenance costs in supporting persistently optimized BLAS". In: 35 (Feb. 2005), pp. 101–121.

[8] UK Mini-App Consortium. *TeaLeaf*. URL: `http://uk-mac.github.io/TeaLeaf/` (visited on 07/28/2018).

[9] R. H. Dennard et al. "Design of ion-implanted MOSFET's with very small physical dimensions". In: *IEEE Journal of Solid-State Circuits* 9.5 (Oct. 1974), pp. 256–268. ISSN: 0018-9200.

[10] S.S. Dosanjh et al. "Exascale design space exploration and co-design". In: *Future Generation Computer Systems* 30 (2014). Special Issue on Extreme Scale Parallel Architectures and Systems, Cryptography in Cloud Computing and Recent Advances in Parallel and Distributed Systems, ICPADS 2012 Selected Papers, pp. 46–58. ISSN: 0167-739X.

[11] H. Esmaeilzadeh et al. "Dark Silicon and the End of Multicore Scaling". In: *IEEE Micro* 32.3 (May 2012), pp. 122–134. ISSN: 0272-1732.

[12] T. L. Falch and A. C. Elster. "ImageCL: An image processing language for performance portability on heterogeneous systems". In: *2016 International Conference on High Performance Computing Simulation (HPCS)*. July 2016, pp. 562–569.

[13] T. L. Falch and A. C. Elster. "Register Caching for Stencil Computations on GPUs". In: *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. Sept. 2014, pp. 479–486.

[14] Thomas L. Falch and Anne C. Elster. "Machine learning-based autotuning for enhanced performance portability of OpenCL applications". In: *John Wiley & Sons* (2016).

[15] Thomas Løfsgaard Falch. "ImageCL and Other Techniques and Tools for Optimizing Applications Utilizing Heterogeneous Computing". In: (2018). ISSN: 1503-8181.

[16] C++ Foundation. *Standard C++*. URL: `https://isocpp.org/` (visited on 07/24/2018).

[17] M. Frigo and S. G. Johnson. "The Design and Implementation of FFTW3". In: *Proceedings of the IEEE* 93.2 (Feb. 2005), pp. 216–231. ISSN: 0018-9219.

[18] Martin Gardner. *The fantastic combinations of John Conway's new solitaire game "life"*. 1970. URL: `https://web.archive.org/web/20090603015231/http://ddi.cs.uni-potsdam.de/HyFISCH/Produzieren/lis_projekt/proj_gamelife/ConwayScientificAmerican.htm` (visited on 12/11/2017).

[19] Al Geist and Sudip Dosanjh. "IESP Exascale Challenge: Co-Design of Architectures and Algorithms". In: *The International Journal of High Performance Computing Applications* 23.4 (2009), pp. 401–402. eprint: `https://doi.org/10.1177/1094342009347766`.

[20] Khronos OpenCL Working Group. *The OpenCL Specification*. URL: `https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_API.pdf` (visited on 07/28/2018).

[21] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 012383872X, 9780123838728.

[22] Michael A. Heroux et al. "Improving Performance via Mini-applications". In: (Sept. 2009).

[23] I. Karlin et al. "Exploring Traditional and Emerging Parallel Programming Models Using a Proxy Application". In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. May 2013, pp. 919–932.

[24] Christoph Kessler et al. "Programmability and Performance Portability Aspects of Heterogeneous Multi-/Manycore Systems". In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE '12. Dresden, Germany: EDA Consortium, 2012, pp. 1403–1408. ISBN: 978-3-9810801-8-6.

[25] Khronos. *OpenCL API 1.0 Quick Reference Card*. URL: `https://www.khronos.org/files/opencl-quick-reference-card.pdf` (visited on 07/28/2018).

[26] Khronos. *OpenCL - The open standard for parallel programming of heterogeneous systems*. 2017. URL: `https://www.khronos.org/opencl/` (visited on 12/09/2017).

[27] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors. A Hands-on Approach*. Morgan Kaufmann Publishers, 2010. ISBN: 978-0-12-381472-2.

[28] Sameer Kulkarni and John Cavazos. "Mitigating the Compiler Optimization Phase-Ordering Problem using Machine Learning". In: (2012).

[29] Mark Harris Luke Durant Oliver Giroux and Nick Stam. *Inside Volta: The World's Most Advanced Data Center GPU*. 2017. URL: `https://devblogs.nvidia.com/parallelforall/inside-volta/` (visited on 12/09/2017).

[30] Mantevo. *Mantevo Project*. URL: `https://mantevo.org/` (visited on 05/03/2018).

[31] Martineau Matthew, McIntosh-Smith Simon, and Gaudin Wayne. "Assessing the performance portability of modern parallel programming models using TeaLeaf". In: *Concurrency and Computation: Practice and Experience* 29.15 (). e4117 cpe.4117, e4117. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4117`.

[32] MPI. *MPI Forum*. 2017. URL: `http://mpi-forum.org/` (visited on 12/14/2017).

[33] G. S. Murthy et al. "Optimal loop unrolling for GPGPU programs". In: *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. Apr. 2010, pp. 1–11.

[34] NVIDIA. *CUDA C Best Practices Guide*. 2017. URL: `http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf` (visited on 12/09/2017).

[35] NVIDIA. *CUDA C Programming Guide*. 2017. URL: `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf` (visited on 12/09/2017).

[36] NVIDIA. *CUDA C Programming Guide - Hardware Implementation*. 2017. URL: `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#hardware-implementation` (visited on 12/09/2017).

[37] NVIDIA. *CUDA Zone*. 2017. URL: `https://developer.nvidia.com/cuda-zone` (visited on 12/09/2017).

[38] NVIDIA. *NVIDIA CUDA Architecture*. 2009. URL: `http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf` (visited on 12/09/2017).

[39] NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*. 2012. URL: `https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf` (visited on 12/12/2017).

[40] NVIDIA. *Register Cache: Caching for Warp-Centric CUDA Programs*. 2017. URL: `https://devblogs.nvidia.com/parallelforall/register-cache-warp-cuda/` (visited on 12/12/2017).

[41] OpenMP. *OpenMP: Enabling HPC since 1997*. 2017. URL: `http://www.openmp.org/` (visited on 12/14/2017).

[42] J. D. Owens et al. "GPU Computing". In: *Proceedings of the IEEE* 96.5 (May 2008), pp. 879–899. ISSN: 0018-9219.

[43] S.J. Pennycook et al. "An investigation of the performance portability of OpenCL". In: *Journal of Parallel and Distributed Computing* 73.11 (2013). Novel architectures for high-performance computing, pp. 1439–1450. ISSN: 0743-7315.

[44] Even O. Rogstadkjærnet. *Parameter Based Optimizations and Code Generation of CUDA for ImageCL*. Tech. rep. Norwegian University of Science and Technology, Department of Computer Science, 2018.

[45] ROSE. *ROSE compiler infrastructure*. 2017. URL: `http://rosecompiler.org/?page_id=31` (visited on 12/12/2017).

[46] Sean Rul et al. "An experimental study on performance portability of OpenCL kernels". eng. In: *Application Accelerators in High Performance Computing, 2010 Symposium, Papers*. Knoxville, TN, USA, 2010, p. 3.

[47] Jason Sanders and Edward Kandrot. *Cuda By Example. An Introduction to General-Purpose GPU Programming*. Addison Wesley, 2011. ISBN: 978-0-13-138768-3.

[48] "Special issue-mpi-a message-passing interface standard". In: *The international journal of high performance computing applications.* 8.3-4 (1994). ISSN: 1094-3420.

[49] Top500. *Top500 List - November 2017*. URL: `https://www.top500.org/list/2017/11/?page=1` (visited on 05/03/2018).

[50] *UPC*. URL: `https://upc-lang.org/` (visited on 07/24/2018).

[51] C. T. Vaughan and R. F. Barrett. "Enabling Tractable Exploration of the Performance of Adaptive Mesh Refinement". In: *2015 IEEE International Conference on Cluster Computing*. Sept. 2015, pp. 746–752.

[52] O. Villa et al. "Scaling the Power Wall: A Path to Exascale". In: *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*. Nov. 2014, pp. 830–841.

[53] L. Wang and K. Skadron. "Implications of the Power Wall: Dim Cores and Reconfigurable Logic". In: *IEEE Micro* 33.5 (Sept. 2013), pp. 40–48. ISSN: 0272-1732.

[54] WG5. *WG5 Fortran Standards Home*. 2018. URL: `https://wg5-fortran.org/` (visited on 07/24/2018).

[55] Wm. A. Wulf and Sally A. McKee. “Hitting the Memory Wall: Implications of the Obvious”. In: (Dec. 1994).

Appendix A

ImageCL Kernels

This appendix shows the ImageCL kernels used to generate the GPU implementations. Listing A.1 shows the first kernel version, which uses a single read-write buffer. Listing A.2 shows the second kernel version, which uses two separate buffers for reading and writing. The last kernel version can be seen in Listing A.3 where two separate buffers have been preprocessed to become three-dimensional, before launching the kernel.

```
#pragma imcl grid(input)
#pragma imcl boundary_cond(input:none)
void kernel_v1(
        int var,
        int num_iter,
        int num_vars,
        Block<double> input
) {
    int X = W + 2;
    int Y = H + 2;
    int Z = D + 2;

    for (int i = 0; i < num_iter; ++i) {
        int READ = (i * (num_vars + 1) + (var + 1)) *
           X * Y * Z;
        int WRITE = (i * (num_vars + 1) + var) * X * Y
            * Z;
        // 5d -> 3d
        Block<double> read = input + READ;
        Block<double> write = input + WRITE;
        write[idz + 1][idy + 1][idx + 1] =
                (read[idz][idy + 1][idx + 1]
                 + read[idz + 1][idy][idx + 1]
                 + read[idz + 1][idy + 1][idx]
                 + read[idz + 1][idy + 1][idx + 1]
                 + read[idz + 1][idy + 1][idx + 2]
                 + read[idz + 1][idy + 2][idx + 1]
                 + read[idz + 2][idy + 1][idx + 1]) /
                   7.0;
    }
}
```

Listing A.1: ImageCL kernel using a single read-write buffer.

```
#pragma imcl grid(input)
#pragma imcl boundary_cond(input:none,output:none)
void stencil_kernel(
        int var,
        int num_iter,
        int num_vars,
        Block<double> input,
        Block<double> output
) {
    int X = W + 2;
    int Y = H + 2;
    int Z = D + 2;

    for (int i = 0; i < num_iter; ++i) {
        int OFFSET = (i * num_vars + var) * X * Y * Z;
        // 5d -> 3d
        Block<double> read = input + OFFSET;
        Block<double> write = output + OFFSET;
        write[idz + 1][idy + 1][idx + 1] =
                (read[idz][idy + 1][idx + 1]
                 + read[idz + 1][idy][idx + 1]
                 + read[idz + 1][idy + 1][idx]
                 + read[idz + 1][idy + 1][idx + 1]
                 + read[idz + 1][idy + 1][idx + 2]
                 + read[idz + 1][idy + 2][idx + 1]
                 + read[idz + 2][idy + 1][idx + 1]) /
                    7.0;
    }
}
```

Listing A.2: ImageCL kernel using read and write buffers.

```
#pragma imcl grid(input)
#pragma imcl boundary_cond(input:none,output:none)
void stencil_kernel(Block<double> input, Block<double>
    output) {
    output[idz + 1][idy + 1][idx + 1] =
            (input[idz][idy + 1][idx + 1]
             + input[idz + 1][idy][idx + 1]
             + input[idz + 1][idy + 1][idx]
             + input[idz + 1][idy + 1][idx + 1]
             + input[idz + 1][idy + 1][idx + 2]
             + input[idz + 1][idy + 2][idx + 1]
             + input[idz + 2][idy + 1][idx + 1]) /
                7.0;
}
```

Listing A.3: ImageCL kernel using 3D read and write buffers.

Appendix B

MiniAMR configuration

This appendix lists the miniAMR configurations used in all of the performance measurements.

Variable	Value	Description
max_blocks	500	Maximum number of blocks per core.
target_active	0	Target number of blocks per core.
target_max	0	Max number of blocks per core, none if 0.
target_min	0	Min number of blocks per core, none if 0.
num_refine	5	Number of levels of refinement.
uniform_refine	0	Says if grid is uniformly refined.
num_vars	40	Number of variables.
comm_vars	0	Number of variables to communicate together.
init_x	1	Initial blocks in x.
init_y	1	Initial blocks in y.
init_z	1	Initial blocks in z.
reorder	1	Ordering of blocks.
npx	1	Number of processors in x.
npy	1	Number of processors in y.
npz	1	Number of processors in z.
inbalance	0	Percentage inbalance to trigger inbalance.
refine_freq	5	Frequency of checking for refinement.
report_diffusion	0	Print checksums.
error_tol	8	Error tolerance.
num_tstep	20	Number of timesteps.
stages_per_ts	20	Number of comm/calc stages per timestep.
checksum_freq	5	Number of stages between checksums.
stencil	7	The 3D stencil used in the application.
report_perf	12	Determines how the performance output is displayed.
num_objects	0	Number of objects to cause refinement.
lb_opt	1	Load balance strategy.
block_change	0	Number of levels a block can change during refinement.
code	0	Decides the way communication is done.
permute	0	Permute communication directions.

Appendix C

Detailed Measurements

This appendix contains the raw data of the measurements presented in Section 5.2. Only the arithmetic average of the ten runs are included, for spatial reasons. See the miniAMR repository for the full raw data (github.com/acelster/miniAMR-NTNU/tree/master/timings).

Table C.1: The time (in seconds) spent inside the kernel code for the GTX 980 system. Timed using CUDA events. This data is illustrated in Figure 5.1(a).

Dimension	V1	V2	V3	V4	V5	V6
4x4x4	0.1061974	0.1108634	0.1051717	0.1210734	0.0974294	0.0838950
6x6x6	0.1218507	0.1270070	0.1202561	0.1356205	0.1406664	0.0942338
8x8x8	0.1392570	0.1477185	0.1399117	0.1507227	0.1986906	0.1047901
10x10x10	0.1386269	0.1454769	0.1360454	0.1496569	0.1990679	0.1024575
12x12x12	0.1394752	0.1489263	0.1409269	0.1498694	0.2334984	0.1036866
14x14x14	0.1456470	0.1590377	0.1504081	0.1548491	0.2670944	0.1051968
16x16x16	0.1499147	0.1651615	0.1609697	0.1608329	0.2567063	0.1078939
18x18x18	0.1728905	0.1783648	0.1683628	0.1782899	0.3047006	0.1144595
20x20x20	0.1909822	0.1956702	0.1851905	0.1840544		0.1233045
22x22x22	0.2137591	0.2230922	0.2053426	0.1991883		0.1354989
24x24x24	0.2348671	0.2443217	0.2254226	0.2109556		0.1419998
26x26x26	0.3036092	0.3040961	0.2958563	0.2931324		0.1789419
28x28x28	0.3372883	0.3391280	0.3364153	0.3204051		0.1894729
30x30x30	0.3565744	0.3609692	0.3585366	0.3443436		0.1969773
32x32x32	0.4026501	0.4041561	0.3998154	0.3810412		0.2044937
34x34x34	0.5272903	0.5337558	0.4897159	0.4609266		0.2679372
36x36x36	0.5727812	0.5777270	0.5499766	0.5003141		0.2775454
38x38x38	0.6246976	0.6291934	0.5981121	0.5406501		0.3036420
40x40x40	0.6910444	0.6902175	0.6728867	0.5803164		0.3233443

Table C.2: The time (in seconds) spent inside the kernel code for the P100 system. Timed using CUDA events. This data is illustrated in Figure 5.1(b).

Dimension	V1	V2	V3	V4	V5	V6
4x4x4	0.1166437	0.1227836	0.1150214	0.1309654	0.1379375	0.1103772
6x6x6	0.1277542	0.1307216	0.1235603	0.1363111	0.2418323	0.1173744
8x8x8	0.1420829	0.1427832	0.1348538	0.1437100	0.3538216	0.1341173
10x10x10	0.1477853	0.1466485	0.1404781	0.1444263	0.3693241	0.1320887
12x12x12	0.1435619	0.1451578	0.1362687	0.1421053	0.4447030	0.1332779
14x14x14	0.1601354	0.1630178	0.1569736	0.1462247	0.4740064	0.1331279
16x16x16	0.1483892	0.1468794	0.1398344	0.1450868	0.4677716	0.1365028
18x18x18	0.1543039	0.1606053	0.1542430	0.1519983	0.4019007	0.1333985
20x20x20	0.1480257	0.1531520	0.1455478	0.1513628		0.1338726
22x22x22	0.1626224	0.1636581	0.1554278	0.1536130		0.1319988
24x24x24	0.1546301	0.1545855	0.1477293	0.1527811		0.1330630
26x26x26	0.1735204	0.1744864	0.1675933	0.1726117		0.1445608
28x28x28	0.1752554	0.1750622	0.1684807	0.1746800		0.1502477
30x30x30	0.2111156	0.2119427	0.2028306	0.1844332		0.1593006
32x32x32	0.1982069	0.2014424	0.1979672	0.1853100		0.1697543
34x34x34	0.2391220	0.2423729	0.2326834	0.2127113		0.1954952
36x36x36	0.2318688	0.2414904	0.2322636	0.2146593		0.1966274
38x38x38	0.2545041	0.2575003	0.2534143	0.2253803		0.2065210
40x40x40	0.2598647	0.2688978	0.2662055	0.2322329		0.2147521

Table C.3: The time (in seconds) spent inside the kernel code for the Titan V system. Timed using CUDA events. This data is illustrated in Figure 5.1(c).

Dimension	V1	V2	V3	V4	V5	V6
4x4x4	0.0908810	0.0885952	0.0872567	0.0966920	0.1037951	0.0870584
6x6x6	0.1054009	0.1038311	0.1030711	0.1080627	0.1758340	0.0969641
8x8x8	0.1114681	0.1088039	0.1089164	0.1124034	0.3057212	0.1174013
10x10x10	0.1260615	0.1237923	0.1238774	0.1171562	0.3115122	0.1191019
12x12x12	0.1164151	0.1143616	0.1127612	0.1141005	0.3648112	0.1178188
14x14x14	0.1490839	0.1461536	0.1460513	0.1204987	0.4116916	0.1188200
16x16x16	0.1161462	0.1143656	0.1135985	0.1147032	0.3817482	0.1194535
18x18x18	0.1262849	0.1263914	0.1265314	0.1234341	0.3383555	0.1169401
20x20x20	0.1162607	0.1180996	0.1178136	0.1207130		0.1175696
22x22x22	0.1289404	0.1316057	0.1317002	0.1250653		0.1167402
24x24x24	0.1222692	0.1205330	0.1211075	0.1228009		0.1168292
26x26x26	0.1479583	0.1444754	0.1473139	0.1435783		0.1274650
28x28x28	0.1389086	0.1369609	0.1367504	0.1418159		0.1263544
30x30x30	0.1687065	0.1664265	0.1658333	0.1492498		0.1279490
32x32x32	0.1457430	0.1442082	0.1438383	0.1472749		0.1262304
34x34x34	0.1780226	0.1770515	0.1773054	0.1791588		0.1717657
36x36x36	0.1748233	0.1739698	0.1731763	0.1779531		0.1700728
38x38x38	0.1958229	0.1912567	0.1934327	0.1906930		0.1756694
40x40x40	0.1954816	0.1942714	0.1936103	0.1972794		0.1779412

Table C.4: The time (in seconds) spent executing the kernel including local kernel setup for the GTX 980 system. Timed using `gettimeofday`. This data is illustrated in Figure 5.2(a).

Dimension	C	V1	V2	V3	V4	V5	V6
4x4x4	0.0073703	0.2081242	0.2118660	0.2068463	0.2238996	0.2820727	0.3328470
6x6x6	0.0210109	0.2270257	0.2353711	0.2291369	0.2436944	0.3310593	0.3462568
8x8x8	0.0480917	0.2494613	0.2593109	0.2491805	0.2625594	0.3943532	0.3634994
10x10x10	0.0905686	0.2559133	0.2668641	0.2575423	0.2717111	0.4041998	0.3692608
12x12x12	0.1572666	0.2669049	0.2785173	0.2708491	0.2807038	0.4486343	0.3797861
14x14x14	0.2457970	0.2822001	0.3004972	0.2937104	0.2983973	0.4984256	0.3972800
16x16x16	0.3753409	0.2859605	0.3121996	0.3082497	0.3081389	0.4972803	0.4078935
18x18x18	0.5315191	0.3083200	0.3275739	0.3172493	0.3291450	0.5480275	0.4146855
20x20x20	0.7224314	0.3255647	0.3499914	0.3403879	0.3386635		0.4292981
22x22x22	0.9650161	0.3479582	0.3799752	0.3621813	0.3571752		0.4454437
24x24x24	1.2519179	0.3704891	0.4058625	0.3866663	0.3740132		0.4587729
26x26x26	1.6174835	0.4583664	0.4869680	0.4795645	0.4800933		0.5254990
28x28x28	2.0127618	0.4977551	0.5385713	0.5365091	0.5187654		0.5604016
30x30x30	2.4692618	0.5116715	0.5684372	0.5664110	0.5530204		0.5810851
32x32x32	0.9086122	0.5683215	0.6291159	0.6248752	0.6048406		0.6129396
34x34x34	1.1976246	0.6771368	0.7556059	0.7124933	0.6810675		0.6826539
36x36x36	1.4837581	0.7479053	0.8249452	0.7980594	0.7573599		0.7288652
38x38x38	1.6446018	0.8001740	0.8925623	0.8624114	0.8126679		0.7761413
40x40x40	2.0072713	0.8585216	0.9631206	0.9440207	0.8619408		0.8451177

Table C.5: The time (in seconds) spent executing the kernel including local kernel setup for the P100 system. Timed using `gettimeofday`. This data is illustrated in Figure 5.2(b).

Dimension	C	V1	V2	V3	V4	V5	V6
4x4x4	0.0071456	0.2671025	0.2727811	0.2642829	0.2809989	0.3865605	0.4823210
6x6x6	0.0139907	0.2798548	0.2816424	0.2736362	0.2871442	0.4909853	0.4876631
8x8x8	0.0270585	0.2989224	0.3003437	0.2930993	0.3026662	0.6072225	0.5088120
10x10x10	0.0462193	0.3139148	0.3116046	0.3057042	0.3105854	0.6303127	0.5137467
12x12x12	0.0809359	0.3195933	0.3225343	0.3127105	0.3181598	0.7177167	0.5272990
14x14x14	0.1280987	0.3408385	0.3453213	0.3406622	0.3248881	0.7564628	0.5305763
16x16x16	0.1832504	0.3344715	0.3308681	0.3249262	0.3277785	0.7531809	0.5398625
18x18x18	0.2441697	0.3384162	0.3481105	0.3405690	0.3337716	0.6955128	0.5358365
20x20x20	0.3246280	0.3311530	0.3382487	0.3307332	0.3303348		0.5371471
22x22x22	0.4269684	0.3451077	0.3536196	0.3445110	0.3349421		0.5347858
24x24x24	0.5588275	0.3379662	0.3435629	0.3368790	0.3359506		0.5413256
26x26x26	0.7226737	0.3578006	0.3709305	0.3639207	0.3605558		0.5606690
28x28x28	0.9078824	0.3572905	0.3693994	0.3638510	0.3576767		0.5665889
30x30x30	1.1005116	0.3932410	0.4125518	0.4037597	0.3681672		0.5884762
32x32x32	1.3168181	0.3809139	0.4053562	0.4022619	0.3684446		0.5974746
34x34x34	1.5161530	0.4249544	0.4491026	0.4398970	0.4083485		0.6247036
36x36x36	1.7730238	0.4137065	0.4506969	0.4429883	0.4151867		0.6318488
38x38x38	2.0799294	0.4382242	0.4699428	0.4666812	0.4371312		0.6461939
40x40x40	2.4213505	0.4445710	0.4880144	0.4843347	0.4435121		0.6610986

Table C.6: The time (in seconds) spent executing the kernel including local kernel setup for the Titan V system. Timed using `gettimeofday`. This data is illustrated in Figure 5.2(c).

Dimension	C	V1	V2	V3	V4	V5	V6
4x4x4	0.0071469	0.2064748	0.2034880	0.2023367	0.2123279	0.2979234	0.3303050
6x6x6	0.0183271	0.2239098	0.2245836	0.2219888	0.2277155	0.3726914	0.3444294
8x8x8	0.0420576	0.2338515	0.2344040	0.2341491	0.2380183	0.5091398	0.3685883
10x10x10	0.0783598	0.2562999	0.2556642	0.2566696	0.2509261	0.5207345	0.3797446
12x12x12	0.1358254	0.2572223	0.2563438	0.2539629	0.2581794	0.5848671	0.3908121
14x14x14	0.2134536	0.2972661	0.2970061	0.2970354	0.2712796	0.6391701	0.3998630
16x16x16	0.3265693	0.2656060	0.2638690	0.2644658	0.2653630	0.6097080	0.4011998
18x18x18	0.4617963	0.2704283	0.2745919	0.2738092	0.2722839	0.5676539	0.3952386
20x20x20	0.6260329	0.2611140	0.2691931	0.2697901	0.2705818		0.4007618
22x22x22	0.8418971	0.2728631	0.2842027	0.2836327	0.2752356		0.4014337
24x24x24	1.0826010	0.2652754	0.2728533	0.2738014	0.2751054		0.4040582
26x26x26	1.3765596	0.2898726	0.2976837	0.2989621	0.2937024		0.4192808
28x28x28	1.7108144	0.2823528	0.2939268	0.2939667	0.2968198		0.4231993
30x30x30	2.0950463	0.3133921	0.3273391	0.3266779	0.3086467		0.4311947
32x32x32	0.7466317	0.2896276	0.3076126	0.3075373	0.3091314		0.4373822
34x34x34	0.9582400	0.3208619	0.3460146	0.3458825	0.3469605		0.4889652
36x36x36	1.0886971	0.3196867	0.3404915	0.3409132	0.3489927		0.4967387
38x38x38	1.2723567	0.3430800	0.3639252	0.3671757	0.3662318		0.5118298
40x40x40	1.4644969	0.3410479	0.3712090	0.3739654	0.3774949		0.5213330

Table C.7: The total time (in seconds) spent on the GPU setup on the GTX 980 system. Timed using `gettimeofday`. This data is illustrated in Figure 5.3(a).

Dimension	C	V1	V2	V3	V4	V5	V6
4x4x4	0.0073703	0.2302281	0.2324934	0.2333764	0.2356438	0.3167083	0.4764558
6x6x6	0.0210109	0.2462970	0.2529744	0.2540843	0.2531504	0.3357991	0.4900289
8x8x8	0.0480917	0.2762884	0.2812823	0.2785908	0.2827151	0.3651879	0.5174810
10x10x10	0.0905686	0.3191121	0.4037814	0.4034254	0.4055604	0.4896621	0.6326700
12x12x12	0.1572666	0.3800141	0.4607580	0.4641017	0.4654644	0.5458656	0.6856358
14x14x14	0.2457970	0.4618743	0.5500196	0.5490368	0.5590005	0.6419319	0.7834983
16x16x16	0.3753409	0.5559618	0.6671284	0.6691950	0.6716441	0.7634433	0.9190428
18x18x18	0.5315191	0.6791231	0.8058579	0.8051728	0.8201209	0.9286292	1.0729010
20x20x20	0.7224314	0.8355320	0.9887194	0.9881465	0.9970952		1.2680958
22x22x22	0.9650161	1.0271266	1.2085781	1.2068704	1.2195026		1.5194769
24x24x24	1.2519179	1.2643665	1.4648295	1.4607497	1.4847478		1.8042998
26x26x26	1.6174835	1.5746065	1.8177953	1.8178682	1.8438357		2.2001473
28x28x28	2.0127618	1.9238926	2.2128598	2.2248059	2.2515476		2.6615578
30x30x30	2.4692618	2.2740041	2.6234000	2.6278242	2.6599501		3.1139864
32x32x32	0.9086122	2.7185443	3.1203432	3.1186156	3.1605476		3.6654812
34x34x34	1.1976246	3.1691453	3.6357149	3.6312770	3.6888896		4.2487024
36x36x36	1.4837581	3.7660000	4.3013449	4.2983524	4.3731541		4.9798939
38x38x38	1.6446018	4.3334171	4.9559952	4.9528446	5.0317539		5.7270977
40x40x40	2.0072713	4.9761028	5.6960418	5.6922564	5.7887588		6.5845832

Table C.8: The total time (in seconds) spent on the GPU setup on the P100 system. Timed using `gettimeofday`. This data is illustrated in Figure 5.3(b).

Dimension	C	V1	V2	V3	V4	V5	V6
4x4x4	0.0071456	1.8264522	1.8340605	1.8213003	1.8299755	1.9317001	2.1630334
6x6x6	0.0139907	1.8695537	1.8768538	1.8762138	1.8838337	1.9781816	2.2160599
8x8x8	0.0270585	1.9311443	1.9291361	1.9345039	1.9526346	2.0228075	2.2619151
10x10x10	0.0462193	2.0277928	2.0033878	2.0148305	2.0258435	2.1113110	2.3342285
12x12x12	0.0809359	2.1249402	2.1310169	2.1310399	2.1406825	2.2277862	2.4548227
14x14x14	0.1280987	2.2608963	2.3743276	2.3850733	2.3834985	2.4801978	2.7031890
16x16x16	0.1832504	2.4475711	2.5444849	2.5459366	2.5668912	2.6536040	2.8932909
18x18x18	0.2441697	2.6706853	2.7940799	2.8014384	2.8132317	2.9055764	3.1555878
20x20x20	0.3246280	2.7422016	2.8578604	2.8499536	2.8886802		3.2290779
22x22x22	0.4269684	3.0338654	3.1793754	3.1879540	3.2175860		3.5758111
24x24x24	0.5588275	3.2836618	3.4440185	3.4368664	3.4397675		3.7862310
26x26x26	0.7226737	3.5645745	3.7058958	3.7607043	3.7372167		4.1422065
28x28x28	0.9078824	3.8057568	4.0206701	4.0109330	4.1202450		4.5245502
30x30x30	1.1005116	4.3052786	4.4807026	4.4667491	4.5942738		4.9477346
32x32x32	1.3168181	4.7854149	4.9769931	4.9559154	5.1206666		5.4364093
34x34x34	1.5161530	5.2498894	5.6827381	5.6198085	5.7138347		6.2233648
36x36x36	1.7730238	6.0837066	6.4510372	6.3354010	6.5953072		6.9304145
38x38x38	2.0799294	7.2209625	7.5475852	7.5704792	7.6699239		8.0947063
40x40x40	2.4213505	8.4141759	8.8534643	8.8954579	8.9236302		9.5720951

Table C.9: The total time (in seconds) spent on the GPU setup on the Titan V system. Timed using `gettimeofday`. This data is illustrated in Figure 5.3(c).

Dimension	C	V1	V2	V3	V4	V5	V6
4x4x4	0.0071469	0.3429077	0.3465342	0.3482060	0.3483514	0.4275089	0.5678922
6x6x6	0.0183271	0.3588242	0.3696713	0.3642248	0.3643409	0.4430144	0.5849208
8x8x8	0.0420576	0.3836883	0.3918675	0.3916831	0.3936455	0.4729691	0.6092271
10x10x10	0.0783598	0.4238001	0.4301665	0.4321546	0.4328987	0.5070485	0.6494425
12x12x12	0.1358254	0.4803811	0.4844067	0.4844861	0.4880177	0.5664367	0.7070891
14x14x14	0.2134536	0.5503432	0.6436904	0.6448545	0.6465440	0.7225932	0.8605801
16x16x16	0.3265693	0.6380166	0.7265569	0.7308560	0.7314997	0.8082942	0.9470014
18x18x18	0.4617963	0.7453130	0.8417395	0.8386361	0.8468661	0.9223022	1.0540252
20x20x20	0.6260329	0.8903324	0.9822825	0.9879142	0.9890780		1.1992608
22x22x22	0.8418971	1.0757827	1.1668479	1.1678145	1.1737764		1.3863890
24x24x24	1.0826010	1.2809139	1.3711645	1.3757898	1.3849594		1.6003110
26x26x26	1.3765596	1.5273060	1.6193900	1.6190686	1.6378478		1.8491346
28x28x28	1.7108144	1.8267236	1.9240256	1.9231333	1.9424857		2.1516878
30x30x30	2.0950463	2.1593729	2.2451233	2.2538738	2.2842046		2.4815136
32x32x32	0.7466317	2.5397175	2.6311318	2.6304901	2.6625194		2.8744011
34x34x34	0.9582400	2.9679766	3.0681203	3.0640880	3.1007073		3.3017180
36x36x36	1.0886971	3.4593233	3.5573029	3.5659821	3.6047801		3.7960155
38x38x38	1.2723567	3.9838573	4.0852571	4.0847859	4.1299290		4.3261018
40x40x40	1.4644969	4.5647094	4.6760043	4.6856599	4.7208786		4.9091783

Table C.10: The total runtime (in seconds) of miniAMR on the GTX 980 system. This data is illustrated in Figure 5.4(a).

Dimension	C	V1	V2	V3	V4	V5	V6
4x4x4	0.0115466	0.8809360	0.8877901	0.8829867	0.9018547	0.9616443	1.1116599
6x6x6	0.0280079	0.9135751	0.9267997	0.9226547	0.9358241	1.0263999	1.1388767
8x8x8	0.0601873	0.9659280	0.9793556	0.9697811	0.9848341	1.1170720	1.1812254
10x10x10	0.1096739	1.0142223	1.1077178	1.0971735	1.1123914	1.2488504	1.2990810
12x12x12	0.1887775	1.0870050	1.1787078	1.1737684	1.1843035	1.3502967	1.3616407
14x14x14	0.2839608	1.1867059	1.2845783	1.2740608	1.2939284	1.4877002	1.4725979
16x16x16	0.4262380	1.2939114	1.4222687	1.4213814	1.4219597	1.6138709	1.6237927
18x18x18	0.5968940	1.4528371	1.5852639	1.5748496	1.5986660	1.8363633	1.7980156
20x20x20	0.8047155	1.6434175	1.8031413	1.7912749	1.7978799		2.0172456
22x22x22	1.0659039	1.8739608	2.0686227	2.0492610	2.0567041		2.2984840
24x24x24	1.3842287	2.1919513	2.3907771	2.3658683	2.3792461		2.6257732
26x26x26	1.8476188	2.7200299	2.9464744	2.9406240	2.9665632		3.2431100
28x28x28	2.4802116	3.4656056	3.7256778	3.7540565	3.7599670		4.0949349
30x30x30	3.0355162	3.7528277	4.0972699	4.1009219	4.1120702		4.4567685
32x32x32	1.7320281	4.4491659	4.8377148	4.8303393	4.8466614		5.1806820
34x34x34	2.2318266	5.2194276	5.6732964	5.6030986	5.6464389		5.9447277
36x36x36	2.5990858	5.8766445	6.3858879	6.3523581	6.3994400		6.8747380
38x38x38	3.2553698	6.9494708	7.5106133	7.5118852	7.5236900		7.9940844
40x40x40	3.9985497	8.1721890	8.8654142	8.8394337	8.8614475		9.1838864

Table C.11: The total runtime (in seconds) of miniAMR on the P100 system. This data is illustrated in Figure 5.4(b).

Dimension	C	V1	V2	V3	V4	V5	V6
4x4x4	0.0137187	2.8538679	2.8672723	2.8460864	2.8735801	2.9871273	3.1990792
6x6x6	0.0233139	2.9053674	2.9210799	2.9090219	2.9329020	3.1368121	3.2588918
8x8x8	0.0451448	3.0116815	3.0171176	3.0099348	3.0389741	3.3277094	3.3580152
10x10x10	0.0757326	3.1226693	3.1004552	3.1062549	3.1273533	3.4407385	3.4342894
12x12x12	0.1236834	3.2302327	3.2451650	3.2327101	3.2498966	3.6437935	3.5673112
14x14x14	0.1854915	3.3946576	3.5186486	3.5239024	3.5087113	3.9423544	3.8325205
16x16x16	0.2612625	3.5928783	3.6937810	3.6883062	3.7127674	4.1249460	4.0468318
18x18x18	0.3411147	3.8365915	3.9689185	3.9673453	3.9810339	4.3313064	4.3179383
20x20x20	0.4439659	3.9146631	4.0482505	4.0275878	4.0751294		4.4097545
22x22x22	0.5703615	4.2416092	4.3941064	4.3993445	4.4190167		4.7746103
24x24x24	0.7387762	4.5115255	4.6837094	4.6649969	4.6756842		5.0144987
26x26x26	0.9508214	4.8512405	4.9895207	5.0409950	5.0268265		5.4185730
28x28x28	1.2112336	5.1382681	5.3658590	5.3492476	5.4607481		5.8578979
30x30x30	1.4586768	5.7366072	5.9067809	5.8937979	6.0094413		6.3496273
32x32x32	1.8008305	6.3065687	6.5116732	6.4652791	6.6211515		6.9361873
34x34x34	2.1002008	6.8840438	7.3229657	7.2702101	7.3573114		7.8550899
36x36x36	2.3261354	7.6871534	8.0757734	7.9344467	8.1656047		8.4778313
38x38x38	2.8393832	9.0778750	9.4489300	9.3957293	9.4734748		9.8732639
40x40x40	3.5083451	10.6808974	11.1469674	11.1330907	11.1843673		11.7061869

Table C.12: The total runtime (in seconds) of miniAMR on the Titan V system. This data is illustrated in Figure 5.4(c).

Dimension	C	V1	V2	V3	V4	V5	V6
4x4x4	0.0110675	1.2388721	1.2417507	1.2434177	1.2495652	1.3422574	1.4681908
6x6x6	0.0244418	1.2746815	1.2866132	1.2766235	1.2814750	1.4283106	1.4975939
8x8x8	0.0528259	1.3062737	1.3144569	1.3130359	1.3191511	1.5983748	1.5459501
10x10x10	0.0951413	1.3654608	1.3737046	1.3763364	1.3701578	1.6376006	1.5912190
12x12x12	0.1623947	1.4229887	1.4241089	1.4212723	1.4246324	1.7647519	1.6563760
14x14x14	0.2470833	1.5284096	1.6203695	1.6227533	1.5972804	1.9802296	1.8258640
16x16x16	0.3714735	1.5939305	1.6785775	1.6870288	1.6867810	2.0435098	1.9240505
18x18x18	0.5196708	1.7231039	1.8210592	1.8154370	1.8221838	2.1265288	2.0387402
20x20x20	0.6981318	1.8707184	1.9665439	1.9785511	1.9767652		2.1971636
22x22x22	0.9303132	2.0960418	2.1925547	2.1928230	2.1916443		2.4009678
24x24x24	1.1985078	2.3229040	2.4136581	2.4262948	2.4318572		2.6451698
26x26x26	1.5779913	2.6626630	2.7544365	2.7578091	2.7713012		2.9803280
28x28x28	2.1293158	3.2782083	3.3729332	3.3660702	3.4034369		3.6407455
30x30x30	2.5857135	3.5276467	3.6154670	3.6178339	3.6404916		3.8574884
32x32x32	1.4926106	4.0405521	4.1450303	4.1393341	4.1723999		4.3923506
34x34x34	1.9062272	4.6456335	4.8268387	4.8259430	4.8480370		4.9878480
36x36x36	2.0860162	5.1586028	5.2506851	5.2680255	5.3200812		5.5980418
38x38x38	2.6940155	6.0179444	6.1350766	6.1375471	6.1859674		6.3938252
40x40x40	3.1714338	6.9754667	7.0533002	7.1277742	7.1530160		7.2315641

Table C.13: The amount of global memory allocated (in Bytes) on the GPU. This data is illustrated in Figure 5.5.

Dimension	V1	V2	V3	V4	V5	V6
4x4x4	28364800	55321600	55321600	56704000	55321600	55408000
6x6x6	67200000	131097600	131097600	134374400	131097600	131302400
8x8x8	131225600	256025600	256025600	262425600	256025600	256425600
10x10x10	226918400	442572800	442572800	453632000	442572800	443264000
12x12x12	360217600	702668800	702668800	720230400	702668800	703766400
14x14x14	537600000	1048780800	1048780800	1074995200	1048780800	1050419200
16x16x16	765363200	1493196800	1493196800	1530521600	1493196800	1495529600
18x18x18	1050291200	2048691200	2048691200	2099891200	2048691200	2051891200
20x20x20	1397708800	2726579200	2726579200	2794726400		2730838400
22x22x22	1814400000	3539635200	3539635200	3628108800		3545164800
24x24x24	2306662400	4500147200	4500147200	4612633600		4507177600
26x26x26	2881740800	5621350400	5621350400	5761843200		5630131200
28x28x28	3544038400	6913638400	6913638400	7086438400		6924438400
30x30x30	4300800000	8390246400	8390246400	8599961600		8403353600
32x32x32	5158323200	10063462400	10063462400	10315008000		10079184000
34x34x34	6124467200	11947136000	11947136000	12245734400		11965798400
36x36x36	7202406400	14050432000	14050432000	14401612800		14072380800
38x38x38	8400000000	16387200000	16387200000	16796800000		16412800000
40x40x40	9723545600	18969728000	18969728000	19443891200		18999363200

Table C.14: The amount of memory (in Bytes) copied from host to device and from device to host. This data is illustrated in Figure 5.6.

Dimension	V1	V2	V3	V4	V5	V6
4x4x4	56704000	82969600	82969600	85043200	110617600	86425600
6x6x6	134374400	196633600	196633600	201548800	262169600	204825600
8x8x8	262425600	384025600	384025600	393625600	512025600	400025600
10x10x10	453632000	663756800	663756800	680345600	884940800	691404800
12x12x12	720230400	1053900800	1053900800	1080243200	1405132800	1097804800
14x14x14	1074995200	1573068800	1573068800	1612390400	2097356800	1638604800
16x16x16	1530521600	2239692800	2239692800	2295680000	2986188800	2333004800
18x18x18	2099891200	3072691200	3072691200	3149491200	4096691200	3200691200
20x20x20	2794726400	4089523200	4089523200	4191744000		4259891200
22x22x22	3628108800	5309107200	5309107200	5441817600		5530291200
24x24x24	4612633600	6749875200	6749875200	6918604800		7031091200
26x26x26	5761843200	8431206400	8431206400	8641945600		8782438400
28x28x28	7086438400	10369638400	10369638400	10628838400		10801638400
30x30x30	8599961600	12584550400	12584550400	12899123200		13108838400
32x32x32	10315008000	15094374400	15094374400	15471692800		15723238400
34x34x34	12245734400	17919104000	17919104000	18367001600		18665600000
36x36x36	14401612800	21074048000	21074048000	21600819200		21952000000
38x38x38	16796800000	24579200000	24579200000	25193600000		25603200000
40x40x40	19443891200	28452992000	28452992000	29164236800		29638400000