Exploring the Optimization Space of Multi-Core Architectures with OpenCL Benchmarks

Deepak Mathews Panickal

Master of Science
Computer Science
School of Informatics
University of Edinburgh
2011

Abstract

Open Computing Language (OpenCL) is an open standard for writing portable software for heterogeneous architectures such as Central Processing Units (CPUs) and Graphic Processing Units (GPUs). Programs written in OpenCL are functionally portable across architectures. However, due to the architectural differences, OpenCL does not warrant performance portability. As previous research shows, different architectures are sensitive to different optimization parameters. A parameter which exhibits good performance on an architecture might not be so for another.

In this thesis, the optimization space of multi-core architectures is explored by running OpenCL benchmarks. The benchmarks are run for all possible combinations of optimization parameters. Exploring the optimization space is not a trivial task as there are various factors, such as the number of threads, the vectorization factor, etc., which impact the performance. The value range that each parameter takes is quite large. For e.g., the number of threads can vary from from 1 to 2^{25}. Four different architectures are evaluated in this thesis. Considering all the parameter combinations for all the four architectures, the optimization space is prohibitively large to be explored within the time constraints of the project. Impossible combinations are pruned to reduce the exploration space.

Over 600,000 runs of the OpenCL benchmarks are executed to exhaustively explore this space and successfully identify the optimal optimization parameters. In addition, the rationality for a parameter being the best on a particular architecture is sought out. The findings of the thesis could be used by developers for significantly improving the performance of their OpenCL applications. They could also be incorporated into a compiler for automatic optimization based on the target architecture.

Acknowledgements

First of all, I wish to sincerely thank my supervisor, Dr. Christophe Dubach, for his extensive guidance and the invaluable advice he gave me throughout the project. I would also like to thank Dominik Grewe, for guiding me and providing me with help on technical issues. Their continuous involvement and ardent interest in the project domain were among the major contributing factors to the success of this project.

Thanks also to everyone who took the time to read through my thesis and give me honest feedback (Christophe, Dominik and Joseph).

And last, but certainly not the least, I am thankful to my parents and friends who have always stood by me.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Deepak Mathews Panickal*)

To my parents and friends who made me who I am today.

Table of Contents

1 Introduction **1**
1.1 Motivation . . . 2
1.1.1 General-Purpose Programming . . . 2
1.1.2 GPU Programming . . . 3
1.2 Contributions . . . 3
1.3 Thesis Outline . . . 4
1.4 Summary . . . 5

2 Background **6**
2.1 Parallel Computing . . . 6
2.1.1 Flynn's Taxonomy . . . 7
2.1.2 Levels of Parallelism . . . 8
2.1.3 Processor Architectures . . . 9
2.2 The Era of General-Purpose GPU Computing . . . 10
2.2.1 Evolution of GPUs . . . 11
2.2.2 General-Purpose Computation on GPUs . . . 11
2.3 Open Computing Language(OpenCL) . . . 13
2.3.1 Platform Model . . . 13
2.3.2 Execution Model . . . 14
2.3.3 Memory Model . . . 17
2.4 OpenCL Mapping to Processor Architectures . . . 19
2.4.1 CPU vs GPU . . . 19
2.4.2 Mapping Parallelism . . . 20
2.5 Summary . . . 21

3 Related Work **22**
3.1 Optimization of CUDA programs . . . 22

3.2 CPU-GPU Performance Evaluation 24
3.3 Benchmarks 24
3.4 Performance Portability of OpenCL 25
3.5 Summary 26

4 Kernel Design **27**
4.1 Optimization Space 27
4.2 The "Empty" Kernel 28
4.2.1 Implementation 28
4.3 The "Vector-Add" Kernel 29
4.3.1 Implementation 30
4.3.2 Coalesced and Non-Coalesced Memory Access 30
4.3.3 Vectorizing the Kernel 31
4.4 The "Compute-Adaptable" Kernel 32
4.4.1 Implementation 32
4.5 Summary 34

5 Experimental Setup and Methodology **35**
5.1 Experimental Setup 35
5.1.1 Compute Device Architectures 35
5.1.2 Test Environment 37
5.2 Methodology 38
5.2.1 Kernel Execution 38
5.2.2 Collecting Execution Times 39
5.2.3 Modelling Graphs 40
5.3 Summary 40

6 Results and Critical Analysis **42**
6.1 Overheads of Thread Creation 42
6.2 Configuring Global and Local Work Sizes 43
6.2.1 On the GPU 45
6.2.2 On the CPU 46
6.3 Examining Memory Access Methods 47
6.3.1 On the GPU 49
6.3.2 On the CPU 50
6.4 Vectorizing the Kernels 50

6.4.1 On the GPU 51
6.4.2 On the CPU 52
6.4.3 Global and Local Work Sizes with Vectorization 52
6.4.3.1 On the GPU 52
6.4.3.2 On the CPU 53
6.4.4 Memory Access Methods with Vectorization 54
6.4.4.1 On the GPU 54
6.4.4.2 On the CPU 54
6.5 Evaluating the "Compute-Adaptable" Kernel 55
6.5.1 On the GPU 56
6.5.2 On the CPU 57
6.6 Summary 58

7 Conclusion 59
7.1 Contributions 59
7.2 Difficulties Encountered 62
7.3 Future Work 63
7.4 Summary 64

A Kernel Implementation 66
A.1 Coalesced Vector-Add 66
A.2 Non-Coalesced Vector-Add 67
A.3 Non-Coalesced Compute-Adaptable Vector-Add 68

B Experiment Results 69
B.1 Vectorization - Memory Access Methods 69
B.2 Vectorization - Global and Local Work Sizes 70
B.3 Compute-Adaptable Kernel 73
B.4 Unrolling Loops 74

Bibliography 76

Chapter 1

Introduction

The need for better performance and parallelism has now led to *heterogeneous computing*, involving GPUs and other highly parallel multi-core architectures. The high complexity of programming for GPUs and the inherent difficulty in adapting general-purpose code for graphics API had for a long time, deterred developers from making use of GPUs for parallel computing. However, the interest towards *GPGPU programming* increased tremendously with the introduction of CUDA (Compute Unified Device Architecture) by NVIDIA in 2007, which allowed software developers to easily develop GPU computing applications in a C-based language.

In 2009, the Khronos[24] consortium introduced OpenCL[50], a programming standard which supports programs that execute across heterogeneous platforms including CPUs and GPUs. Most CPUs and GPUs available in the market are now OpenCL compliant. The key merit of OpenCL is that it allows programmers to write code which is vendor-independent. It provides easy-to-use abstractions and a broad set of programming APIs which are based on the C language.

OpenCL guarantees *functional portability* but not *performance portability*. Though OpenCL compliant, each architecture is designed according to specifications decided by its manufacturer. This creates the problem of the same program exhibiting poor performance on hardware with similar technical capabilities. The hardware can be from different manufacturers or even different generations of the same model. Functional portability is necessary but ensuring performance portability is also essential from a developer's point of view.

If an application written in portable code is not fast enough to be usable on a platform, then developers will prefer to program the application in the platform's native language. However, the application might have to be written more than once for the

application to work optimally on multiple platforms. As this adds to undue overhead for the developers, a more feasible solution will be to optimise the application individually for each platform. This method could be applied for exploring the OpenCL optimization space. Since programs are guaranteed to be portable, developers could tune an OpenCL program which was meant for one architecture and make it optimized for another without losing correctness.

This thesis explores the optimization space of various multi-core architectures, by running OpenCL benchmarks on them with exhaustive combinations of optimization parameters. The effects of the experiments are observed and the rationality behind the observed results are also sought out. This aids in understanding the architecture in more detail, and also helps in interpreting the interactions between optimization parameters. Moreover, the exploration is done on both memory-bound and compute-bound benchmarks to identify the optimal configurations for each scenario.

1.1 Motivation

Most of GPUs and CPUs that are now manufactured and deployed around the world are OpenCL compliant. This creates an exciting possibility of *write-once* and *run on multiple* hardware (rather than being variants of the same type of hardware, multiple hardware in this case are as diverse as CPUs and GPUs) which is always appealing to programmers. However, OpenCL only provides functional portability. This means that the code written and optimized for one device will run correctly on another OpenCL compliant device, although not necessarily with peak performance.

There are a multitude of architectural variations to be considered while developing applications. Since this has to be done for each architecture separately, it becomes a *time-consuming* task for the programmer to develop programs which exhibit optimal performance.

1.1.1 General-Purpose Programming

In programming for CPUs, the programmer need not be aware about the low level details such as the cache sizes, the memory region where the program is running and so forth. CPUs are general-purpose made which perform well for a *wide range of applications* including sequential and parallel applications. A CPU might have various cache memory hierarchies for speeding up of execution, but all of these details are

abstracted from the programmer.

1.1.2 GPU Programming

In programming for GPUs, the programmer needs to select many low-level details such as the memory sizes, the number of threads to be created and also has to consider whether the memory accesses are coalesced. Non-coalesced memory accesses itself, could cause a reduction in performance[26]. This occurs when instructions in the same cycle access different locations within the same bank. In programming for CPUs, unrolling loops is not usually done by the programmer as it is taken care of by the compiler. Compilers for GPUs are relatively new compared to the ones for CPUs and most of the process of optimising the program is left to the programmer.

1.2 Contributions

There are two main contributions of this project:

The main contribution is identifying the *optimal optimization parameters* which fit each architecture by exhaustively exploring the optimization space. This is done by investigating the effect of applying all combinations of optimization parameters using OpenCL benchmarks. Four multi-core architectures are evaluated, namely, Intel, Nvidia, and ATI architectures. Considering all the parameter combinations, the optimization space is *prohibitively large* to be explored within the time constraints of the project. Over 600,000 runs of the OpenCL benchmarks are executed for checking all possible combinations. Impossible combinations are pruned to reduce the number of experiments required to explore the space.

Some of the parameters explored in this thesis are, the method of memory access implemented in the benchmarks, the total number of threads created for the execution, etc. The optimal configuration for the parameters contributes to significant improvement in performance. The total number of threads being a parameter itself exemplifies that. On some architectures, the best performance is achieved when the total number of threads is equal to the number of processors whereas on other architectures, the total number of threads has to be orders of magnitude larger for optimal performance.

In addition to identifying the best combinations of parameters, the *rationality* for a parameter being the best or the worst on a particular architecture is sought out. The results of the experiments are critically analysed to provide a deeper understanding of

the underlying architecture and to realize the interactions between optimization parameters. Ultimately, this provides the potential for extending the results of this thesis to further exploration of the optimization space with some other parameters.

Along with the aforementioned contributions, further gains from this thesis are:

1. Programmers working with OpenCL will have a better understanding of how to develop programs with *optimal performance*.

2. The identified parameters can be later incorporated into a compiler which will then *automatically apply* the specific optimizations based on the target architecture.

1.3 Thesis Outline

This thesis is organized into seven chapters including this chapter. The organization is as follows–

- **Chapter 2** gives a background perspective of the concepts and terminologies used throughout this thesis. Parallel computing, general-purpose GPU computing and OpenCL are some of the concepts which are discussed.

- **Chapter 3** presents the related work. Prior work in exploration of optimization space, benchmarks used, etc., are discussed with respect to the work done in this thesis.

- **Chapter 4** discusses the kernel design. The different kernel versions that are designed, the reasons behind the design decisions and the kernel code implementations are also provided.

- **Chapter 5** gives an overview of the environment for the experiments. The methodology for conducting the experiments and evaluating the results are also discussed.

- **Chapter 6** presents the experiment results. The various graphs modelled for different kernel versions are exhibited. Outcomes of the experiments and observations are also addressed in this chapter.

- **Chapter 7** finally concludes the thesis by presenting the contributions, discussing the difficulties faced during the project, and suggesting the future work.

1.4 Summary

This chapter has introduced the thesis, giving an overview of the performance portability issue in OpenCL. The motivation for the core idea of exploring the optimization space of multi-core architectures is described. Furthermore, the contributions of this project are also listed. The next chapter outlines various technologies and concepts such as parallel computing, general-purpose GPU computing, OpenCL model, etc., used throughout this thesis.

Chapter 2

Background

This chapter discusses the various technologies and concepts used in this thesis. The first section introduces parallel computing, discussing the various terms and concepts. Section 2.2 then describes the emergence of general-purpose GPU computing and section 2.3 follows by presenting the OpenCL standard and explaining the different OpenCL models. Finally, the mapping of OpenCL to various processor architectures is provided, including the comparison of CPU and GPU architectures.

2.1 Parallel Computing

Parallel computing has always been a candidate for high performance computing, strongly securing its place in computation-intensive areas such as scientific research, weather forecasting, etc. However, as far as the consumer sector was concerned, up until a few years ago the single-core microprocessor dominated the market. This was possible since the processor performance could be improved upon by increasing the processor clock frequency. It is only befitting to quote *Moore's law*, which has now stayed valid for the past 45 years, and is even used to set targets for research and development in the semiconductor industry.

> *"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year... Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years."*
>
> – Gordon E. Moore, Intel Co-founder[31]

According to Moores Law, the number of transistors on a chip roughly doubles every two years. This law has stayed valid over the years by cramming more and more transistors into the same core. As frequency scaling began to reach its limits due to physical constraints such as power consumption and heat generation, it became impractical and the focus slowly shifted to parallel computing as the dominant paradigm[2].

Parallel computing is a seemingly easy approach to exploit computing resources and build more powerful systems as the basic idea is to scale up the system, i.e., to add more computing power as needed. Dual-core and quad-core processors have become the norm nowadays. Though it may be trivial to think, that now there are four processors instead of just the one before, the main impact of this paradigm shift is in the rise of new forms of computing such as *general-purpose GPU computing* and more recently, *heterogeneous computing*. In heterogeneous computing, tasks are executed in parallel on CPUs and GPUs obtaining unprecedented levels of performance. The following section presents more about various concepts of parallel computing.

2.1.1 Flynn's Taxonomy

According to Flynn's taxonomy[12], architectures are classified based on the presence of single or multiple streams of instructions and data. There are four classifications as listed in the table 2.1. The descriptions are provided below.

SISD An architecture in which a single processor executes a single instruction to operate on data stored in a single memory.

SIMD An architecture in which multiple processing elements execute the same operation on multiple data simultaneously.

MISD An architecture in which multiple processing elements perform different operations on the same data. This can be seen in a pipeline architecture where the same data moves along a pipeline and different operations are performed on it.

MIMD An architecture in which different processing elements perform different operations on different pieces of data. The data can be stored in a shared memory or a distributed memory.

Flynn's Taxonomy		
	Single Instruction	**Multiple Instruction**
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Table 2.1: Flynn's Taxonomy classifies architectures into four categories based on the presence of single or multiple streams of instructions and data.

2.1.2 Levels of Parallelism

In this section, the three levels of parallelism are introduced, namely, instruction-level parallelism, task-level parallelism and data-level parallelism.

Instruction-level Parallelism

In instruction-level parallelism (ILP), more than one instruction is executed during a single clock cycle. Though the program to be executed might be following a sequential execution model, various micro-architectural techniques such as out-of-order execution or pipe-lining can be applied to exploit ILP.

Task-level Parallelism

In task-level parallelism, each processor executes a different thread or process on the same or different data. For e.g., in a dual core processor, two different cores can execute two different threads at the same time. If the threads are part of the same process, the data being worked upon can be the same. Task-parallelism emphasizes on distributing the process or thread across parallel processing nodes.

Data-level Parallelism

In data-level parallelism, each processor executes the same thread or process on different data. For e.g., adding two vectors can be done in a single clock cycle if there are as many processors as the number of additions to be performed. This model is in sync with the SIMD model. Data-parallelism emphasizes on distributing the data across parallel processing nodes.

2.1.3 Processor Architectures

In this section, various processor architectures which form the basis and are implemented in many of the current CPUs and GPUs are presented along with a discussion of their features and shortcomings.

Multi-processor architectures

A 'multi-processor' system, as its name suggests is a single computer system which has multiple processing nodes. Multi-processors can be classified based on their execution model.

Vector processors In vector processors, there are multiple, pipelined functional units which has the capability to execute single instructions on vectors or arrays of data[10]. All the functional units execute the instructions in lock-step fashion on the local data. According to Flynn's taxonomy[12], vector processors follow the SIMD model. Vector processors are very power-efficient as the units consist of simple execution units. There is no instruction checking done at runtime and no other complex features implemented in the processor. Taking the simplicity into account, the space required for the units on the die is also considerably smaller, thereby leading to higher number of units and more power efficiency.

VLIW processors The VLIW architecture takes advantage of ILP, by executing multiple instructions in parallel but the difference being that the schedule of instructions is determined when the program is compiled. It has multiple execution units like vector processors, but it is capable of executing different instructions at the same time. The EPIC[46] architecture which became the basis for the Intel Itanium[47] architecture, has evolved from the VLIW architecture with additional features such as register renaming and predicated execution. The VLIW architecture is more power hungry than vector processors. Unlike super-scalar processors, the schedule of instructions is statically determined by the compiler, rather than by the processor.

Super-scalar processors In super-scalar processors, multiple functional units are available on the processor so that multiple instructions can be executed per clock cycle. Data dependencies between instructions are dynamically checked at runtime for doing this. Super-scalar processors are different from multi-core processors where the redundant units are entire processors and parallelism is achieved

by executing one thread per core. Though super-scalar processors process multiple data items in a single clock, they do not process multiple data items for a single instruction. Super-scalar processors are much more power-hungry than VLIW and vector processors due to their dynamic behaviour. The units are more complex due to added functionalities such as out-of-order execution, branch prediction, etc.

Multi-core processors Multi-core processors contain multiple independent cores on a single chip (also known as chip multiprocessor). Though the cores are independent, they do share some resources such as cache memories, main memory between them. Sharing cache memories aids in exhibiting task-parallelism where the cores can work on the same data simultaneously. In addition, implementing multiple functional units (such as ALUs) in a single core aids in data-parallelism. According to Flynn's taxonomy[12], it follows the MIMD model. Multi-core processors can implement super-scalar or vector architectures or even a hybrid of both for added performance benefits. Most real programs get maximum benefit with a continuum of both data-parallelism and task-parallelism. Most of the processors being manufactured now, try to support a combination of these configurations.

Many-core processors Many-core processors are similar to multi-core processors but with a much higher number of cores. It is not required that all the cores have to be all on a single chip, but all the cores will be in a single processor package. They are designed for a higher degree of parallelism, supporting advanced levels of scalability. Many-core processors follow the MIMD model. They usually consist of simpler elements such as vector processors, whereas multi-core processors usually consist of more complex elements such as super-scalar processors. Each core in many-core processor is simple, small, and independent from each other. Typically, a multi-core processor will have fewer cores (two to six) whereas many-core processors usually have 32 or more cores.

2.2 The Era of General-Purpose GPU Computing

Over the years, GPUs have evolved from being a *configurable graphics processor* to a *programmable many-core multi-threaded processor* with tremendous computational power and very high memory bandwidth. This architectural evolution brought forward

a large increase in number of applicable domains for GPUs. They have now become a powerful platform for computationally demanding tasks in a wide variety of applications.

2.2.1 Evolution of GPUs

Rendering highly complex graphics scenes is inherently a parallel computing problem. The multi-billion dollar gaming market is an area which keeps pushing the processing limits of GPUs to be able to render complex scenes in real-time at interactive frame rates.[32] To solve these problems, GPUs had to evolve to have the capability to execute tens of thousands of parallel threads using hundreds of parallel processors.

The introduction of fully programmable hardware was the vital step for enabling general-purpose computation on GPUs. GPUs initially, did not have support for high precision floating point operations[15] as they were not mandatory in graphics applications. The support for high precision floating point operations and capability to handle data-parallel computational problems enabled researchers to accelerate scientific and visualization applications significantly. With the exposure of computation capability, programmers began to use the tremendous parallel processing power for general-purpose computation.

2.2.2 General-Purpose Computation on GPUs

This section discusses the types of general-purpose computations which benefit from GPUs. The transition from using OpenGL[54] for general-purpose programming on GPUs to CUDA[35] is also presented.

Which types of computations?

GPUs are designed for processing graphics and as mentioned before, this involves computing pixels from processing independent data elements. This computation is done in parallel, which is effectively data-parallelism. Data parallelism and data independence are two key attributes of computer graphics computation[14]. The cost of computation to communication ratio can be termed as *arithmetic intensity*. As the amount of computation gets higher, the arithmetic intensity increases. The computations which benefit from GPU processing are ones with high arithmetic intensity and high amount of data independence (for data-parallelism).

As a result of data-parallelism, there is a lower requirement for sophisticated flow control, and the high amount of computation hides the memory access latencies, all of which contributes to very high throughput for such programs. With high amount of computation, there will be a large number of compute-bound threads which increases the computation-to-communication ratio. This results in more computation per memory access and thus hiding the memory latencies.

OpenGL to CUDA

OpenGL[54] is a powerful cross-platform API developed for writing applications that produce high-quality 2D and 3D computer graphics. It provides a single interface to GPUs developed by different hardware manufacturers. OpenGL provides a large amount of programming constructs which enables a developer to easily write portable graphics applications, which efficiently make use of the graphics pipe-line process.

When GPGPU computing was still in its initial stages, programmers had to use OpenGL to try to express general-purpose programs which was quite difficult as general-purpose programs have nothing to do with graphics. The required calculations had to be expressed in the terms of an image rendering process. Another issue was that the programmable units in the GPUs were accessible only as part of the graphics pipeline. This meant that the program had to go through all the pipeline stages, even though no graphics was involved in them. GPGPU computing became widely popular with the introduction of CUDA[35] parallel computing model in 2006 by Nvidia.

CUDA is a *scalable parallel computing architecture* developed by Nvidia that enables Nvidia GPUs to execute general-purpose programs. Programmers write GPPGU programs in a language 'C for CUDA'. The language being very similar to standard C was very familiar to programmers and helped them to focus on the more important issues of parallelism. With its easy programming model and huge parallel computational potential, CUDA tremendously increased the interest in the GPGPU computing sector. GPGPU computing is now a part of operating systems such as Microsoft Windows 7, Apple Snow Leopard and recently, also on Linux (KGPU[23]). Millions of CUDA-enabled Nvidia GPUs have already been sold, with it being used in a wide variety of domains such as medical imaging[52], molecular dynamics[51], ray tracing[42], and much more.

As can be seen with almost all of the technologies nowadays, an open-standard implementation to CUDA was inevitable. An open-standard for GPGPU computing could aid the processor manufacturers in adapting their hardware to support a common

model. Such an open-standard effort by Apple led to the development of OpenCL, which is presented in detail in the next section. Both CUDA and OpenCL share similar architectural features with CUDA being exclusive to Nvidia architectures, and OpenCL more general.

2.3 Open Computing Language(OpenCL)

OpenCL is an open industry standard maintained by the Khronos Group[24] for writing programs that execute across heterogeneous computing devices such as CPUs, GPUs, and other processors. The OpenCL framework provides a runtime system, libraries, and a programming language which is an extension to the standard C language (based on C99). This helps programmers to develop portable general-purpose software which can take advantage of all the different platforms that support OpenCL.

The OpenCL standard was originally developed by Apple Inc. who first proposed the standard in 2008. It is now managed by the industry consortium - the Khronos Group[24] which includes major CPU, GPU, and software companies (such as Apple, IBM, Nvidia, AMD, Samsung). The latest OpenCL specification (OpenCL v1.1 revision 44) was released on June 1st 2011. In this thesis, the API according to the OpenCL 1.0 specification is used. This is since the new changes as per OpenCL v1.1 specification are not relevant and also proper drivers have not yet been made available for platforms such as Nvidia at the time of running experiments.

The *write once, run anywhere* behaviour of OpenCL is the one major property which sets it apart from other such languages for the GPU. During runtime, the OpenCL code is compiled just-in-time for the particular architecture and hence the programmer need not bother about which target architecture the program will be running on, as long as it supports OpenCL. OpenCL supports both data-parallel and task-parallel programming models, as well as the hybrid of them. Primarily driving the design is the data-parallel model. It also provides easy-to-use low-level hardware abstractions and a broad set of programming APIs using which developers can query and identify the actual device capabilities and create efficient code.

2.3.1 Platform Model

The OpenCL specification defines a platform as a host connected to multiple OpenCL devices which are composed of a number of compute units. Compute units can be

further divided into a number of processing elements. Figure 2.1 illustrates how all of these devices interact together. A brief description for each is provided below.

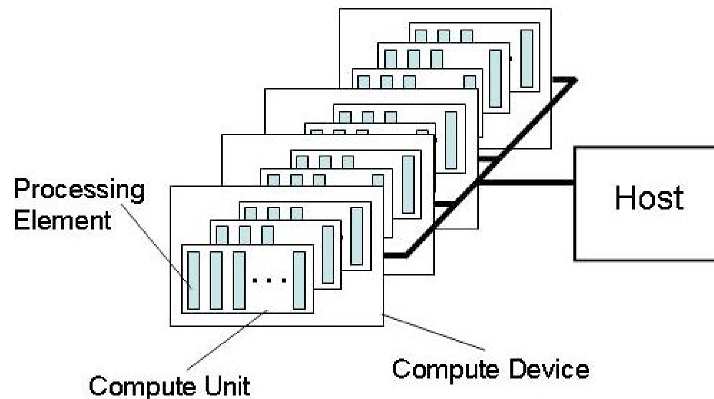


Figure 2.1: The OpenCL Platform Model specifies a host which is usually a CPU connected to multiple OpenCL compute devices such as GPUs or DSPs. The compute devices consists of a collection of compute units(cores) which are further composed of multiple processing elements. Figure from [24].

Host A host usually consists of a CPU and is responsible for running the host application. The host application runs natively on the host and submits commands to the OpenCL device. The commands to be submitted are queued up in a data structure called the command queue which is then scheduled onto the device. Execution of kernels, reading and writing of memory objects are examples of some of the commands which are submitted.

Devices A device can correspond to a multi-core CPU, a GPU, and other processors such as DSPs, etc. A single device is composed of a number of compute units, such as the individual cores in a multi-core CPU.

An aspect to be noted of this model is that, provided the host device also supports OpenCL, programmers can partition a program into serial code and parallel code which are best suited for the CPU and the GPU, respectively. Thereby, the execution can go back and forth between the devices making the best utilization of them.

2.3.2 Execution Model

In this section, the OpenCL execution model is presented. Before talking further about the execution, the various terms used are introduced.

Program An OpenCL program consists of one or more kernels and auxiliary functions which are used by the kernels. Programs are written in an OpenCL-C language. The language has extensions which are for e.g., specifying memory spaces and also additional keywords for specifying a function as a kernel function. The OpenCL compiler which is a part of the runtime, compiles programs to create binaries which can be executed or saved for later loading.

Kernel The kernel is a function in an OpenCL program that is executed on a device. The return types of kernels are always void as all inward and outward communication is done through the memory. All the necessary operations such as copying of memory objects, setting kernel arguments etc., required for the kernel execution are managed by the host application.

Work-Item Instances of the kernel are executed in parallel on the compute units of the device. A work-item or thread, is one such instance of the kernel and is the work performed by one compute unit. So at the same instant, a device with N compute units can only execute N work-items. However in practice, more work-items are scheduled for each compute unit to keep the pipelines full for optimum performance. The same kernel code is executed by all the work items concurrently, but the specific path taken can vary based on the algorithm. Work items are identified in two ways - one is by using a global ID and the second way is through a combination of a local ID and work-group ID, which will be explained in detail in the next segment.

Work Group As mentioned before, a collection of work-items are assigned for execution on a single compute unit. This collection of work-items is called as a work-group. When a kernel is enqueued for execution, two parameters pertaining to work-groups can be specified, which are the global work size and the local work size. The global work size is the total number of kernel instances or work-items that are to be started for computation, whereas the local work size is the number of work-items that are assigned to one work-group. So the number of work-groups will be always equal to the global work size divided by the local work size. If the local work size is not specified, then the OpenCL implementation will decide how to break down the global work-items into appropriate work-groups. In case there are more work-groups than the available number of compute units, the work-groups will be scheduled one by one on the compute

units. A compute unit will always concurrently finish executing the work-items in one work-group before executing work-items from another work-group.

Global, Local and Work-group IDs IDs are provided to the programmer in order to be able to identify the work-item, access the required memory address and make control decisions as needed. In OpenCL, an index space is defined for kernel execution which is called as an NDRange. An NDRange is an N-dimensional index space, where N is one, two or three. In OpenCL 1.0, the starting global and local ID is always (0, 0, 0). Every work-item has a unique global ID which is a point in the index space and which will be ranging from 0 to global work-group size minus one. Similarly, work-groups are also assigned unique IDs.

A Simple Kernel Execution

In OpenCL, the execution model is based on parallel execution of the kernel, with the process involving both the host and the compute device. The steps involved in kernel execution are listed in the table 2.2.

1. Device Setup
Initialize platform
Get devices and create command queue
2. Device and Host Buffer Setup
Create memory buffers on the host and device
Copy input data from the host memory to device memory
3. Kernel Initialization
Load kernel source code from file
Create program object and build the program
4. Kernel Execution
Set kernel arguments
Execute the OpenCL kernel
5. Output copying and Cleanup
Copy output data from device memory to host memory
Delete all allocated objects

Table 2.2: The steps involved in an OpenCL kernel execution.

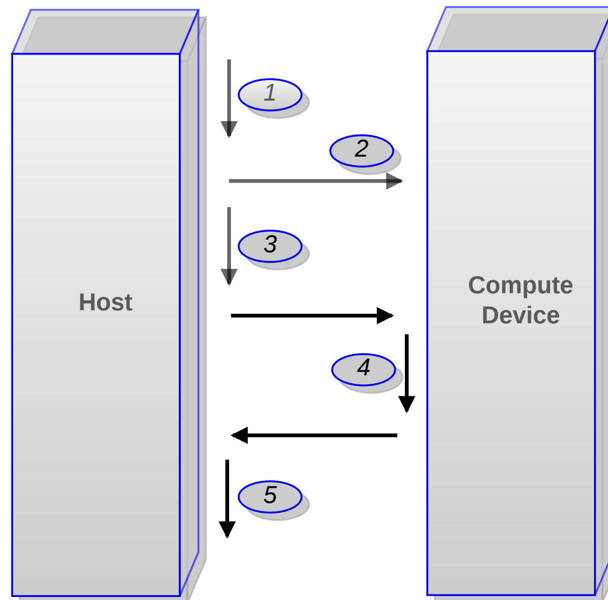


Figure 2.2: The interactions between the host and compute device for a kernel execution. The steps involved are listed in the table 2.2.

The figure 2.2 shows how each of the steps presented in the table 2.2 relate to the host and the compute device. Although the host is required for the initial setup of the execution process, the compute device executes the kernel independent of the host. and so the host can perform other computations in the meantime. The only way for the host and the compute device to communicate is by copying data from the host memory to the device memory and vice versa. Debugging OpenCL programs are therefore quite difficult as the only way to ensure computation is done correctly is to copy the data back to the host and then verify the output.

2.3.3 Memory Model

In this section, the memory model of OpenCL is presented. One important aspect of OpenCL is that it exposes the non-unified memory model of the device. The memory in OpenCL devices is classified into four regions - global, constant, local and private. The task of deciding the region to store data is solely up to the programmer. Figure 2.3 shows the memory hierarchy as defined by OpenCL. A brief description of each of the memory regions is provided below.

Global and Constant memory The global memory region is the main means of communication between the host and the device. The host can create read/write

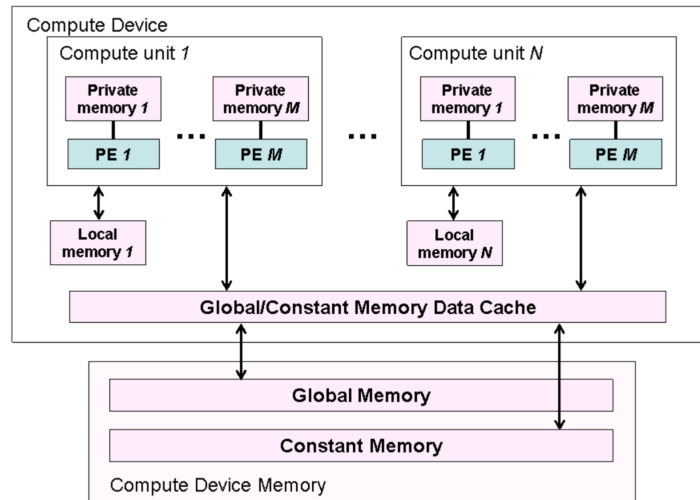


Figure 2.3: The OpenCL Memory Model defines four regions of memory accessible to work-items while executing a kernel. The global memory in the compute device is accessible to the host also. Figure from [24].

buffers on the global memory of the device using commands. It is accessible to all the work-items and a programmer can use the global address space qualifier in a kernel to denote that an object is to be stored in global memory. Though it is usually the largest space available on the device, the access latency is also much larger compared to the other regions. The constant memory is a part of the global memory, but the difference between them is that while the host can read and write into this region, the kernel has read-only access. The constant address space qualifier is used to denote that an object is to be placed in constant memory. To take an example, in AMD GPUs, the constant memory is similar to the constant caches and global memory is similar to the off-chip memory that is available on the GPU.

Local Memory The 'local' memory is named as such as it is the memory that is available only to a local work group. In other words, the local memory is made private to a compute unit. So the compute units local memory will be accessible to all the work-items that are part of the same work-group. The space available is much smaller, but has low latency compared to the global region. Using the local memory, work-items in a work-group can share data among them quickly. The local address space qualifier is used to denote that an object is to be placed in local memory. The local memory is similar to the local data share that is available on the current generation of AMD GPUs.

Private Memory This memory region comes further below in the classification hierarchy and is the region that is accessible to only a single work-item. This is the fastest and the smallest memory that is available to a work-item. The private address space qualifier is used to denote that an object is to be placed in constant memory. Finally, the private memory is akin to the registers in a single CPU core.

2.4 OpenCL Mapping to Processor Architectures

This section discusses the general architectural differences between CPUs and GPUs. The mapping of OpenCL to these processor architectures is also presented with an example for squaring an array of elements.

2.4.1 CPU vs GPU

As mentioned in section 2.2.2, until recently, general-purpose code was run only on CPUs, while GPUs were used only for graphics. This was optimal as CPUs have always been designed for running general-purpose code, which is mostly sequential code, with maximum efficiency. On the other hand, GPUs are designed for running data parallel and computationally intensive programs. Nowadays, GPUs have also become programmable like CPUs, and developers are trying to extract the maximum potential out of both of these architectures by using 'the right processor for the right task'.

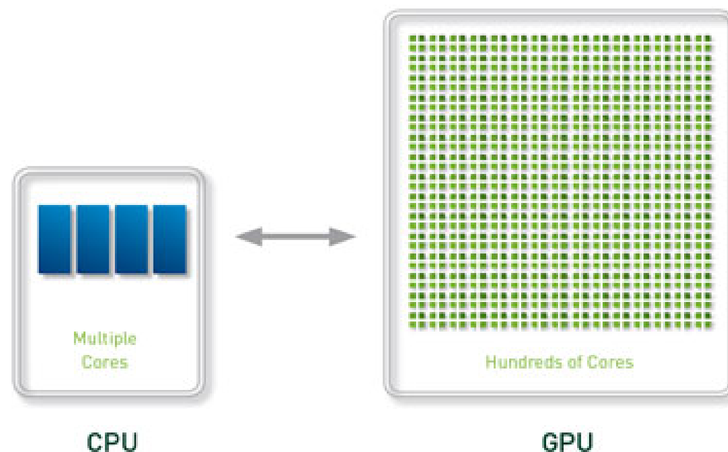


Figure 2.4: CPUs consist of a few number of cores(2 to 6), whereas GPUs consist of hundreds of cores. Figure from [34].

How to know which task is suitable for which processor? For better understanding about this, the key architectural differences between CPU and GPU architectures are examined below.

- **Cache Memories –** Today's CPUs have at least two or three levels of cache, which helps it in increasing the effective bandwidth by minimizing memory accesses. On the other hand, GPUs might not have cache memories and even if they do, they are quite small as they exist primarily to accelerate texture filtering. Another point to note is that CPUs can cache both read and write operations, whereas the GPUs cache only read-only texture data.[41]
- **Clock Frequency –** CPUs usually have a higher clock frequency (ranging from 2000 MHz to 3500 MHz) than GPUs. The higher clock speed helps them to compensate for the lower number of cores. The current generation of GPUs have a clock frequency of around 500 to 1 GHz.
- **Advanced Features –** GPUs usually have vector processors which do not have advanced features such as branch prediction, and out of order execution. This enables GPUs to have much higher number of computational units (see figure 2.4) due to the lesser complexity and the larger space available on the die. CPUs however have such advanced features since they are designed for general-purpose computation.
- **Context Switching –** On GPUs, thread context switching is implemented in hardware, which enables it to switch between thousands of threads very quickly. CPUs depend on the operating system to take care of context switching and this is much slower.

2.4.2 Mapping Parallelism

The difference in sequential programming and parallel programming can be shown through an example. Consider an example in which an array of elements has to be squared. As seen in the code listing 1, the sequential code contains a for-loop which is not present in the OpenCL code.

With the sequential code, the same thread does the computation for all the elements. The sequential code is an example of application usually run on general-purpose CPUs. With the OpenCL code, "n" threads are created which can do the computation in parallel in a single clock cycle. The OpenCL code can be run on supported CPUs and

```c
void square(int n,
            const float *a,
            float *result)
{
int i;
for (i=0; i<n; i++)
        result[i] = a[i] * a[i];
}
```

```c
kernel square (const float *a,
               float *result)
{
int id = get_global_id(0);

result[id] = a[id] * a[id];
}
//executes n workitems
```

Listing 1: Sequential C code is presented on the left and the corresponding OpenCL code on the right. If only one processor is available, the OpenCL code will execute similarly to the C code.

GPUs. Another point to note is that the sequential code takes an additional argument "n" for the size of the array. The "get_global_id()" function call is used to identify the thread, and assuming that there are "n" threads, each thread id will correspond to each element of the array. So the squaring done by each thread will be for each of the elements in the array. Execution of the OpenCL code on a device with a single processing element is similar to that of the sequential code.

2.5 Summary

This chapter has given an overview of various technologies and concepts used throughout this thesis. It introduced parallel computing explaining about different types of parallelism and the prevalent processor architectures such as vector, super-scalar, multi-core, etc. The rise of general-purpose computing on graphics processors with technologies such as CUDA was also described. The chapter also presented the OpenCL standard with further details about the different models. The walk-through of a simple kernel execution was also given. Finally, the mapping of OpenCL to various processor architectures is provided. The next chapter discusses related work.

Chapter 3

Related Work

This chapter presents the prior work and fields of research related to this thesis. The first section discusses about the optimization of CUDA programs. The applications, architectures and evaluation methodologies are described. Section 3.2 then looks at prior work in performance evaluation of CPUs and GPUs and section 3.3 follows with a discussion of various benchmarks. Finally, section 3.4 presents an overview of previous research in performance portability of OpenCL.

3.1 Optimization of CUDA programs

In this project, the main focus is on the optimization of OpenCL programs. More research has been done in the CUDA area, as it is older and more mature than OpenCL. In papers by Ryoo et al.[45, 44], optimization principles and application performance has been investigated using CUDA on the Nvidia GeForce 8800 GTX. Their work focuses towards searching the optimization space and identifying features that contribute to more performance. The optimizations they have considered included intra-thread parallelism, resource-balancing and redistribution of work across thread blocks. Through optimizing and experimenting with *matrix multiplication* and other kernels, they have found that *global memory latency* is a major performance bottleneck and use of local storage and appropriate thread granularity gives a considerable improvement. As the optimization was for CUDA, only one device (Nvidia GeForce 8800 GTX) was evaluated. In this thesis, experiments are run on four devices, namely the Nvidia Tesla C2070, the ATI Mobility Radeon HD 5470, the Intel Core i3-350M, and the Intel Core i7-990X.

They have concluded that even though matrix multiplication is a simple application, there are a significant number of optimization configurations to be considered and only experimentation can determine whether the upsides of an optimization compensates for potential downsides. Another area of research is in porting of CUDA programs to OpenCL after optimization. This have been done by Du et al.[40] but the approach that they explored was to *automatically generate* multiple versions of the same kernel with the same algorithm, but with different configurations of optimization parameters. The best performing kernel is then heuristically selected from all the different versions.

The disadvantage of the *auto-tuning method* is the time cost involved in searching for the best version, since the number of versions will be proportional to the number of different configurations. Du et al.[40], through their experiments found that auto-tuning heuristics is a good method to improve performance, but while designing the algorithm, the architectural features should be taken into account. For OpenMP, which is a common parallel programming framework, Lee et al.[28] devised an automatic translation framework to translate OpenMP code to optimized CUDA code. Other than optimizations such as loop unrolling, source-to-source optimization of CUDA has been investigated by Lionetti et al.[30] on an Nvidia GTX 295 processor. The approach that they adopted was to apply optimizations such as kernel partitioning to reduce register load and using dual GPUs to engage separate threads for better performance, thereby reducing running time from 52.7 sec to 7.9 sec.

An interesting point to note is that CUDA, being a more mature framework compared to OpenCL has better optimizations built into its compiler. This can be attributed to the fact that CUDA compilers are built specifically for Nvidia graphics hardware. However, as Komatsu et al.[25] have evaluated and concluded, the optimizations that are done automatically by the CUDA compiler could be done *by hand* to the OpenCL kernels resulting in comparable performance.

Komatsu et al.[25] applied the same optimizations done by the CUDA compiler, manually, to the PTX code generated by the OpenCL C compiler. Loop unrolling was one of the optimizations they considered and and by applying loop unrolling, they found that the execution time decreased by approximately 67.8%. Carrilo et al.[4] suggested the use of loop splitting and branch splitting as another optimization which has been proved to be very effective on GPUs as they improve occupancy by reducing the register load, but these optimizations can be counter-productive on the CPU.

3.2 CPU-GPU Performance Evaluation

Carr et al.[3] have investigated the differences between CPU and GPU performance on matrix operations. This was done in precedence to devising an efficient three-pass GPU algorithm for rendering subsurface scattering. They have found that the CPU overtakes the GPU for small matrices, but for matrices larger than 2000 elements, the GPU performs better than the CPU. The reason for this behaviour is that the GPU is computation-bound and requires more elements to increase the computation-to-communication ratio.

Similarly, in evaluations by Che et al.[6] and Owens et al.[38], different applications have been implemented for the GPU and the CPU and their performances, compared. Some of the evaluated applications are DES encryption, dynamic programming and game physics. Che et al.[6] have compared the parallel performance of the architectures by implementing code for multi-core CPUs in OpenMP and code for GPUs in CUDA C. They have found that the CUDA version achieved a *35x speed-up* over the multi-core OpenMP version. Performance evaluation of heterogeneous computing where computations are done using the CPU and GPU in parallel, have been analysed by Ohshima et al.[37]. By using a load balancer method for optimal partitioning of computation, they proved that the execution time was reduced to 44.1% for the CPU and 59.5% of that for the GPU.

In most papers, the comparison of performance in CPUs and GPUs usually resulted in the GPU emerging as the winner. Contrasting these results, Lee et al.[29] have investigated the extent of difference in performance between these architectures. They tuned and optimized programs for both CPU and GPU and after experimenting, concluded that though there are claims that GPUs are *100X to 1000X faster* than CPUs, proper tuning makes the GPU only *2.5X faster* than the CPU. This study also shows the importance of proper tuning of parameters and the extent to which it affects the performance.

3.3 Benchmarks

Before beginning the experiments, it has to be decided which are the appropriate benchmarks to be used. Ryoo et al.[45] in their study for CUDA optimization have used matrix multiplication for the initial study. Matrix multiplication is a common benchmark, also used in studies by Larsen et al.[27], Fatahalian et al.[11] and Jiang et

al.[21]. In this thesis, synthetic benchmarks are created which could be adapted for exploring all the optimization parameters. The benchmarks are designed such that the behaviour of the benchmark could be explained considering the underlying device architecture. In the application study, Ryoo et al.[45] have used selected benchmarks from the medical domain such as MRI-Q which are *computationally intensive* and benchmarks such as PNS (Petri Net Simulation) which has *high memory-to-compute ratio*. Their criteria for the selection of applications was to have a large variety of instructions, operate on large data sets and to have more control flow than micro-benchmarks.

In their study towards OpenCL performance portability, Rul et al.[43] have used Parboil[39] benchmarks which included seven CUDA kernels out of which three (CP, MRI-Q and MRI-FHD) were selected and hand-translated to OpenCL for the evaluation. Rodinia[5] is a benchmark suite by Che et al. for evaluating multi-core CPUs and GPU platforms. It included benchmarks such as breadth-first search, similarity-score of websites, etc.

Danalis et al.[8] have recently designed another benchmark suite called The Scalable Heterogeneous Computing (SHOC) Benchmark Suite which is a selection of benchmarks for testing the performance and stability of GPUs and CPUs. It includes implementations in both OpenCL and CUDA. The selected benchmarks include basic parallel algorithms such as FFT, Sort etc.

3.4 Performance Portability of OpenCL

There are few publications that have studied about the performance portability of OpenCL. Recently, Rul et al.[43] have studied the performance portability of OpenCL. The study has been done on four different architectures, including *Intel, Tesla, ATI* and the *IBM Cell* (heterogeneous processor with a general-purpose core and several co-processing elements). For the experiments, three benchmarks and two optimization parameters were used, namely loop unrolling and vectorization. In their study, they have also investigated interactions between the parameters finding that with the addition of vectorization, the optimal loop unrolling factor decreases.

In addition, the paper also discusses the sensitivity of some architectures to the optimization parameters. The ATI FirePro is more sensitive and have different optimal values compared to the Nvidia Tesla. They have concluded that each architecture requires an *exclusive set* of optimizations and the *performance is not portable* across architectures.

However, they have only observed the behaviour of the benchmarks and have not looked into the causes for the behaviour. In this thesis, this has been improved upon by explaining the rationality behind the behaviour of the benchmarks for each architecture. The design of the benchmarks and their implementation are presented in the next chapter.

3.5 Summary

As seen in this chapter, previous research in program optimization of general-purpose GPU computing is limited in the area of OpenCL optimization due to the fact that OpenCL is a relatively new programming framework. Nevertheless, there has been a considerable amount of work done towards optimization of CUDA programs. This chapter also looks at performance evaluation of CPUs and GPUs. As seen, OpenCL is in many ways similar to CUDA and work on optimizing CUDA programs has demonstrated the enormous amount of effort and expertise required in this area. Furthermore, OpenCL is supported on multiple architectures and this makes it harder to make a program perform optimally. The various benchmarks used in both CUDA and OpenCL optimization are discussed. Finally, an overview of previous research in performance portability of OpenCL is also presented. The next chapter describes the design and implementation of various OpenCL kernels used for experiments in this thesis.

Chapter 4

Kernel Design

This chapter discusses the design and implementation of various OpenCL kernels used for experiments in this thesis. The first section describes the optimization space being explored, listing the optimization parameters that are considered. The remaining sections give an overview of the design of kernels. The rationality behind various design decisions is explained. Finally, code walk-throughs and kernel implementations are provided.

4.1 Optimization Space

Though all the devices for the experiments are OpenCL compliant, their architectures can vary greatly within the context of GPUs, and can be as different as a highly parallel GPU and a general-purpose CPU. The exploration of the optimization space has been done by applying each parameter to the source code and implementing different versions. Many of the optimization parameters considered here have been considered in previous research. For running the experiments, various configurations are designed for each parameter. Based on the previous research, the optimizations considered are:

- Configuring the global work size, i.e, the total number of threads that are created in the device for computation;
- Adjusting the local work size or the work-group size;
- Evaluating different memory access methods;
- Vectorizing the code to exploit vector operations supported by the device architecture;

The optimizations mentioned above are also explored on a compute-bound and memory-bound kernel. For this purpose, the computation intensity of a kernel is designed to be configurable, so as to vary the kernel being memory-bound or compute-bound. Loop unrolling is another optimization parameter that is considered, but due to time constraints of the project, only the preliminary analysis has been done. The results are included in the appendix B.4.

4.2 The "Empty" Kernel

One of the optimization parameters is the global work size, or the total number of threads used for computation on the device. For creating a single thread, a device will take some amount of time. Considering the tremendous speeds at which the device functions, the time will certainly be negligible. However, for the experiments, more than a million threads are created for a single kernel execution. The overhead of creation of such a large number of threads could have some influence on the experiment results.

```
1 __kernel void empty()
2 {
3 }
```

Listing 2: The Empty Kernel.

As the name suggests, the idea behind implementing an 'empty' kernel is to use a kernel which was empty, i.e. with no computation at all, so that the execution time of the kernel will be equivalent to the time for creating the threads. In case the overhead is large enough to skew the results for the other experiments, the actual computation values could be approximated by removing the overhead time.

4.2.1 Implementation

For implementing the kernel, a function is written with no arguments and no variables. Listing 2 shows the code for the kernel and it can be seen that no computation is done in the function. The kernel keyword designates the function as an OpenCL kernel.

4.3 The "Vector-Add" Kernel

The main requirements that are identified for choosing an appropriate kernel for doing the optimization process are:

- To implement a kernel which is simple to infer the execution in detail;
- To have the potential to tweak the kernel for applying all the planned optimizations.

The first requirement is necessary since we want to have an understanding of the instructions that are generated by the compiler. This has been very helpful in debugging various difficulties, during the design of compute-bound kernels. The second requirement aids in maintaining the first requirement as we can build upon the existing kernel for assessing more optimization parameters.

```
1  __kernel void vector_add(__global int *A,
2                           __global int *B,
3                           __global int *C)
4  {
5          int i = get_global_id(0);
6          int step = get_global_size(0);
7
8          int start = i;
9          int end = NO_OF_ELEMENTS;
10
11         for(int j = start; j < end; j = j + step)
12                 C[j] = A[j] + B[j];
13 }
```

Listing 3: The Simple Vector-Add Kernel.

In the vector addition kernel, the task is to add two vectors together. The two vectors are provided as input to the kernel and their sum is calculated. An important point to note here is that calculating the sum is not the focus here, but the *amount of computation* done by the device. The computed result is then copied back to the host and verified for correctness. The length of the two input vectors are always the same and are decided based on the device memory capacity, such that the device can hold

the result vector also. In this section, the different versions of the vector-add kernel that has been implemented for evaluating the optimizations are presented.

4.3.1 Implementation

The implementation for the simple vector-add kernel is given in listing 3. A brief walk-through of the code is presented here. The input and the output vectors are passed as arguments (A,B and C) to the kernel function. The *get_global_id()* function call provides the thread identifier and the *get_global_size()* provides the total number of threads that are created. The *NO_OF_ELEMENTS* is a macro which is the number of elements in the vectors. The for-loop ends when the loop count reaches the vector size and in every clock cycle, adjacent elements from the input vectors are summed up by the threads. The memory access method implemented here is coalesced access which will be explained in detail in the next section.

4.3.2 Coalesced and Non-Coalesced Memory Access

Before the kernel execution, the two input vectors are copied into the global memory of the device. In the kernel, this data can be accessed from the memory in two different ways, namely, coalesced access and non-coalesced access. The reason for comparing these two methods is to evaluate which access method is most suitable for which architecture. This variation occurs due to architectural differences such as cache memories, number of cores, etc.

Clock	**Memory Access**								
C_1	T_1	T_2	T_3						
C_2				T_1	T_2	T_3			
C_3							T_1	T_2	T_3

Figure 4.1: The memory access sequences for the coalesced memory access method, for a simple example of three threads accessing an array of nine elements.

Figure 4.1 depicts how coalesced access takes place for each clock cycle. In coalesced access, the data elements which are adjacent to one another are accessed by adjacent but different threads. The threads are adjacent in the sense that they are created one after the other and their identifiers are linear. With coalesced access, as can

be seen from the figure, a block of data from the input vectors can be cached. Since the data required by all the threads in the same clock cycle are present in the cache, there will be no cache-miss and the bandwidth will be optimum.

Clock	Memory Access								
C_1	T_1			T_2			T_3		
C_2		T_1			T_2			T_3	
C_3			T_1			T_2			T_3

Figure 4.2: The memory access sequences for the non-coalesced memory access method, for a simple example of three threads accessing an array of nine elements.

In non-coalesced access, adjacent elements are accessed by the same threads as can be seen in figure 4.2. The adjacent elements are not accessed in the same clock cycle. This behaviour will have implications on the performance based on the device architecture. If the architecture does not support large cache memories, then non-coalesced memory access could result in a large number of cache-misses, thereby degrading the performance. The implementation for the coalesced vector-add kernel is provided in the appendix A.1. For the non-coalesced vector-add kernel, please refer to the appendix A.3.

4.3.3 Vectorizing the Kernel

Another optimization parameter that is explored is vectorization. OpenCL supports vector data types which can be used by developers for extracting more performance from the device. This is possible only if the device supports vector instructions. *float4* is commonly supported by most architectures and the ATI architectures have a natural type for float4. This translates to higher performance as more computation can be done in a single clock cycle. The OpenCL implementation will accept these types in the kernel even if the compute device does not support the vector data types. It is up to the device compiler to convert the data types to the appropriate native instructions.

In the vector-add kernel, four combinations of the int data type have been evaluated. These are *int2, int4, int8* and *int16*. With vectorization, a compute device can execute vector operations in a single clock cycle. For e.g., using int4 in the kernel, the compute device can access four elements of the vector in a single clock cycle. This speeds up

execution more or less by a factor of the vectorization level. The implementation for the vectorized kernel is explained in the code walk-through for the next section.

4.4 The "Compute-Adaptable" Kernel

In real world general-purpose applications, there are both memory-bound and compute-bound applications. The matrix multiplication application which is commonly used as a benchmark, can be adapted to be compute-bound or memory bound as have been done by Jiao et al.[22]. The goal behind developing a compute-adaptable kernel is to be able to control the amount of computation being done by a single thread. By controlling the amount of computation, the optimal configuration can be found for a particular computational intensity.

All the various versions of the kernel that have been designed so far are memory-bound as the amount of "work" done by a single thread is minimal. In other words, the *arithmetic intensity* of the threads is low. Every element from both the input vectors is accessed from the global memory and the result is written back to the global memory. Hence for every one computation, there are three memory accesses taking place. The memory accesses are being done to and fro, the global memory, which has the worst latency of all memory regions in the device.

4.4.1 Implementation

To increase the computational intensity of the kernel, a *configurable parameter* is added. The implementation of this kernel took a greater amount of time than expected due to some unforeseen issues[1]. In order to configure the amount of computation, a for-loop was added around the already existing addition, with the loop-count as the configurable parameter. To get around the memory latency issue, the private memory is used for the input vectors. So instead of directly accessing the global memory three times for each computation, the data is copied to private memory once. Then only the private memory is accessed for the entire duration of the computation. To gain further understanding, the code for the compute-bound kernel is provided in listing 4.

There are two additional arguments to the kernel function, which are the *loop-count* variable for determining the amount of computation and the *loop-increment* variable

[1]The difficulties faced were primarily due to code optimizations issues by the ATI and Nvidia compilers. A bug was also discovered in the ATI compiler while designing this kernel. For more details, please refer to section 7.2.

```
__kernel void vector_add(__global VECTOR_TYPE *A,
                         __global VECTOR_TYPE *B,
                         __global VECTOR_TYPE *C,
                         __global int iter,
                         __global int inc)
{
        int i = get_global_id(0);
        int step = get_global_size(0);
        int start = i;
        int end = NO_OF_ELEMENTS / VECTORIZATION;
        VECTOR_TYPE a,b,val;
        int c = inc;
        for(int j = start; j < end; j = j + step)
        {
                a = A[j];
                b = B[j];
                val = a + b;
                int k = 0;
                for(; k < iter;)
                {
                        val = val + k; k = k + c;
                }
                C[j] = val;
        }
}
```

Listing 4: The Compute-Adaptable Vector-Add Kernel

which always has a value of 1. The reason for passing this as a dynamic parameter to the function is to prevent compiler optimization issues (See section 7.2 for more details). The compute-bound kernel is also vectorized using the *VECTOR_TYPE* macro for the data type and the *VECTORIZATION* macro for controlling the vectorization factor. Line 21 which shows the computation part has been slightly modified from the simple vector-add kernel to make the result dependant on the loop-count. The rest of the code is similar to the simple vector-add kernel. In the listing 4, the coalesced

memory access method is used for accessing the global memory. For the non-coalesced version, please refer to the appendix A.3.

4.5 Summary

This chapter explained in detail about the design and implementation of various OpenCL kernels used for experiments in this thesis. The optimization space explored was described in the first section. The designs of various kernels were discussed in the subsequent sections. These are the empty kernel, the vector-add kernel, the compute-adaptable kernel, and their variations. The empty kernel was designed for investigating thread creation overhead on the architectures.

Different memory access methods, such as coalesced and non-coalesced memory accesses are explained with code-walkthroughs provided for their respective kernel implementations. Vectorizing the kernels which is another optimization is also described. Finally, implementation for the compute-adaptable kernel is provided. The main feature of the compute-adaptable kernel is that the computation intensity of the kernel can be configured through a parameter. This design helped in easily identifying through experiments, the best optimization parameter configuration for a certain level of computation intensity. The next chapter describes the experiment environment and the methodology for performing the experiments.

Chapter 5

Experimental Setup and Methodology

This chapter presents an overview of the experiment environment and the methodology for the experiments. The first section presents the experimental setup with a brief introduction of the architectures of the evaluated devices. The environment and the specification of the test machines are also provided. Section 5.2 describes the methodology, explaining about the steps involved in executing kernels, the collecting of data from the experiments and modelling of graphs for the result analysis.

5.1 Experimental Setup

In this section, the architectures of the four compute devices are presented, along with a comparison of their features. The test machine specifications are also described.

5.1.1 Compute Device Architectures

In this section, the architectural details of the platforms that are used for the experiments are described. There are two GPUs, the Nvidia Tesla C2070[1] which is a high-end GPU and the ATI Mobility Radeon HD 5470 which is a mobile GPU designed for laptops. Similarly, there is a high-end CPU which is the Intel Core i7-990x and a CPU designed for laptops which is the Intel Core i3-350M. Further specification information about each architecture is available in the table 5.1.

[1] Initially, an Nvidia GeForce GTX 580 was selected as the high-end GPU for the experiments. However during the rigorous exercising of the device by the experiments, it was discovered that the device had a faulty memory. It was then replaced with the Nvidia Tesla C2070. For details, please refer 7.2.

Compute Devices				
	ATI 5470[48]	**Nvidia C2070**[49]	**Intel i3**[17]	**Intel i7**[18]
Processing Elements	32	448	2	6
Core Frequency	750MHz	1.15GHz	2.27GHz	3.47GHz
Compute Units	2	56	4	12
Work-group Size	128	1024	1024	1024
Memory	256MB	6GB	4GB	12GB
Bandwidth(GB/s)	25.6	144	17.1	25.6
Performance(GFLOPS)	120	1030	18.08	107.55

Table 5.1: The specifications of the four compute devices.

Intel Core i3-350M and i7-990X

The Intel Core i7-990X is a recent high-end multi-threaded multi-core processor, offering six cores running at a frequency of 3.47GHz. The Intel i3-350M offers only two cores and runs at a much lower frequency of 2.26GHz. Through *Intel Hyper-Threading*[20] technology, both processors can support two threads per core, and thus 12 threads can be run on the i7 and four threads on the i3, at the same time. Both processors have three levels of cache memories, with each core having an L1 cache of 32KB and a 256KB L2 cache. The difference is that there is only 3MB of L3 cache available in the i3 for the two cores, whereas in the i7, the six cores altogether share 12MB of L3 cache. The i7-990X also has the latest *SSE4.2 instruction set* extensions enabling it to support a new range of SIMD instructions.

ATI Mobility Radeon HD 5470

The ATI Mobility Radeon HD 5470 has two compute units, each which contains 16 stream cores. Each stream core within a compute unit executes an instance of a kernel in lockstep. Each of the stream core is a five ALU Very Long Instruction Word (VLIW) processor. With the *VLIW* architecture, each ALU in a stream core is capable of independently executing different instructions. Each compute unit has 32KB of shared memory. Threads are grouped into sets of 64 called wavefronts, and the shared memory usage dictates the number of concurrent wavefronts that can run on one compute unit.

System Specification		
	Machine-One	**Machine-Two**
CPU	Intel Core i3-350@2.27GHz	Intel Core i7-990X@3.47GHz
CPU Processors	4	6
Memory	4 GB DDR2	12 GB DDR2
GPU	ATI Mobility Radeon HD 5470	Nvidia Tesla C2070
GPU Cores	32 cores @ 750MHz	448 cores @ 1.15GHz
GPU Memory	256 MB GDDR5 @ 900MHz	6 GB GDDR5 @ 1.5GHz
Platform	AMD Stream SDK v2.4	AMD Stream SDK v2.4
		Nvidia Computing SDK 4.0.8
OS	Linux Mint 10-2.6.35-22	Ubuntu 11.04-2.6.38-8

Table 5.2: The specifications of the test machines.

Nvidia Tesla C2070

The Nvidia Tesla C2070 is composed of 56 compute units, also called as streaming multiprocessors(SM). Each compute unit has 8 scalar processing units running in lock-step. Thus the number of processing elements or cores is 448. Multi-threading allows hundreds of threads to be run simultaneously and it aids in hiding memory latency. It also has various on-chip memories such as read-only constant caches and shared memory which also help in alleviating memory bandwidth. The shared memory (also called as local data storage) is 64KB which is available to each SM, and this can be partitioned as 16KB of L1 cache and 48KB of shared memory or vice-versa. It also has 768KB of L2 cache shared among all the SMs.

5.1.2 Test Environment

The performance of our kernels have been measured on two test machines,“Machine-One” with the Intel Core i3 as the CPU and ATI Mobility Radeon HD 5470 as the GPU, and “Machine-Two” with the Intel Core i7 as the CPU and Nvidia Tesla C2070 as the GPU. Machine-Two has been provided by the ICSA[19] facility at the University of Edinburgh. The full system specification for the two test machines are provided in the table 5.2.

5.2 Methodology

The same programs are tested on the four architectures to relatively identify the bottlenecks for each architecture. The performance variations depend on the specific chipset and the core frequencies. To ensure that the captured results are not skewed based on overheads such as thread creation time, experiments to are run first to identify the significance of these overheads . The data transfer time is not included in the results since the interest is in investigating whether any architectural specific features are responsible for the performance difference. The data transfer times are logged, to ensure that the experiments are functioning correctly.

As seen in previous research, this process of tweaking the programs and repeatedly running experiments required a lot of effort and was a time-consuming task as the optimization space has to be searched for the best combination of tuning parameters. Considerable amount of time is required to gather sufficient data from the experiments for relevant performance comparisons. An *iterative approach* is used to ensure that results are obtained throughout the process. Each optimization parameter is applied successively, the corresponding kernel executed, data gathered, pre-processed, and graphs modelled. This step-by-step methodology ensured that even if all the optimizations cannot be applied or tested, the project has results. The major phases involved in the experiment process are now presented.

5.2.1 Kernel Execution

Different kernels have been designed for exploring different optimization parameters. For more details about their implementation, please refer to chapter 4. The kernels that have been implemented are listed in the table 5.3.

For executing the kernels, *test execution programs* have been written in C, which loaded the kernels, built and executed them. For each type mentioned in table 5.3, i.e. empty, memory-bound and compute-bound, a program is written for executing the kernel. The test execution programs took various command line arguments as input through which various optimization parameters such as the global work size, local work size, etc., could be configured. Along with the optimization parameters, the total vector size and the type of device (whether the CPU or the GPU) could also be selected.

Before executing the kernel, the device memory where the output will be stored is reset. In some cases, when the kernel is given for execution to the device, the computation might not be done due to some program error, but the execution will be successful.

Kernels		
None	**Memory-Bound**	**compute-bound**
empty	vector_add	computation_coalesced_vd
	coalesced_vd	computation_noncoalesced_vd
	noncoalesced_vd	
	vectorized_coalesced_vd	
	vectorized_noncoalesced_vd	

Table 5.3: The list of kernels used as benchmarks.

In such a case, the output data that is copied from the device memory will be the data from a previous execution. Resetting the memory before each execution prevents this error. The program processes the optimization parameters provided as command-line arguments and then follows the steps for executing a kernel. These steps are mentioned in detail in section 2.3.2. The kernel is executed after setting the parameters as required.

The execution time and transfer times are recorded using *profiling information* available from the device. The loaded kernels are each executed ten times and all the execution times are recorded. Executing each kernel ten times reduces the chance of outliers or randomness in the collected data. The output data from the executed kernel is copied back to the host memory from the device memory. Tests are implemented in the test execution programs to check whether the computations are performed accurately.

5.2.2 Collecting Execution Times

In total, over *600,000 runs* have been performed on the four devices to gather all the data, taking over a total of around *30 hours* for the execution. There are eight different kernels having four to five configurable parameters each. The parameters take a large range of values. For e.g., the number of threads vary from 1 to 2^{25} and the vectorization factor varies from 1 to 16. Robust test automation has been designed for exploring such a large optimization space, collecting the data, verifying computation results and error case handling.

For automating the experiments, *automation scripts* have been written in Python. Python has been selected for the purpose as it has been found to be quite proficient

for string parsing and pre-processing purposes. These scripts run the test execution programs with all possible configurations for the optimization parameters. To reduce the exploration space and subsequently the runtime of the scripts, the parameter combinations which are incorrect or irrelevant are pruned. The test execution programs produce a string as output containing information such as the parameter list, execution times for the ten kernel executions, data transfer times, etc.

The scripts process the output from the test execution programs. The output is tested for incorrect executions or runtime errors. The final processed data is stored into CSV files which are then processed by the graph pre-processing scripts for modelling graphs.

5.2.3 Modelling Graphs

The CSV files generated by the automation scripts contain raw data from the experiments. The data is then processed by graph pre-processing scripts which are also written in Python. Considerable amount of analysis is required to identify the relevant data to model the graphs. For e.g., for the compute-bound kernel, the best configuration is identified by analysing the data. The scripts also determine the median of the ten execution times for the graphs.

The graph pre-processing scripts create data files for each different type of graphs. The processing of these data files is done by using *R scripts*. Using R gave the capability to automate the modelling of complex graphs and also the ability to further tweak them as needed. For each type of graph modelled, individual R scripts have been written to manually enhance them.

5.3 Summary

This chapter presented the experiment environment and the methodology for conducting the experiments. Brief descriptions of the architectures of the four compute devices under evaluation are also provided. Two test machines are used for running the experiments. The specification for the test machines is described. In the methodology section, all the various kernels are listed. The steps taken for the execution of kernels and the verification of the computation results are discussed. The collecting and pre-processing of data from the large number of experiments was a time-consuming task. This chapter also describes the various python scripts used in performing the test

automation and the R scripts used for modelling the graphs. The next chapter presents the experiment results and their critical analysis.

Chapter 6

Results and Critical Analysis

This chapter presents the results of the experiments and provides critical analysis of the results. The first section reports the impact of running the empty kernel, to understand the overheads of thread creation. The subsequent sections present the results for all the kernel variants presented in chapter 4. The results of the experiments are reported in terms of graphs. The analysis of the effect of applying each optimization parameter is also given. Finally, the performance of various device architectures are compared and summarized for each of the experiments.

6.1 Overheads of Thread Creation

The overhead in creating a thread for execution is investigated for each architecture. Since millions of threads can be created for a single kernel execution, this overhead can affect the results for the experiments. The execution times for the empty kernel have been collected for all the architectures. As the kernel did not involve any computation, the execution time will be equivalent to the time for creating threads.

Figure 6.1 shows the overhead incurred as the number of threads is increased. For the smaller number of threads, all the architectures perform equally with minimal overhead. For the Intel i7, a slight increase in execution time occurs once the number of threads is more than 1024. Both the ATI and the Intel i3 show similar behaviour initially, taking the same execution time. However, as the number of threads goes beyond one million, the ATI executes faster. The Nvidia behaves similarly with exponential increase in execution time after the number of threads goes past one million. The overhead for the GPUs is less than that for CPU's for large number of threads($>$1 million),

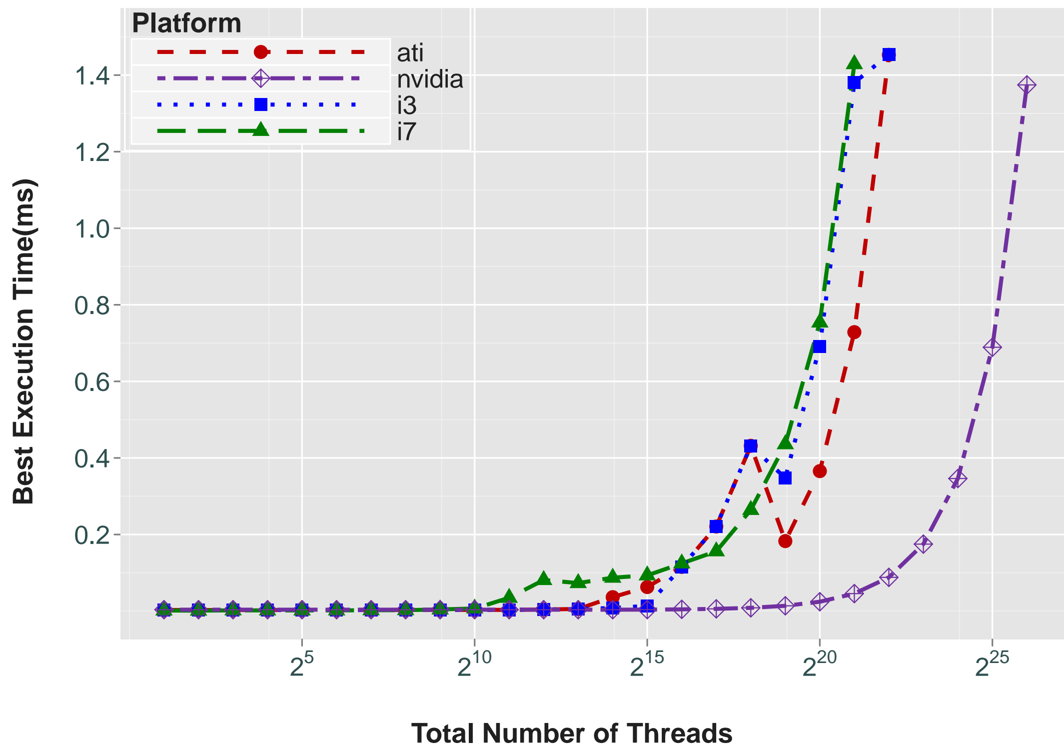


Figure 6.1: The overhead for the creation of threads on the four devices.

. This can be attributed to the higher number of compute units in the GPUs which makes them better at handling the heavier load.

For all the architectures, the graph shows that the execution time is quite negligible for very large number of threads. Even for more than one million threads, the execution time is less than 2 ms. This means that if the computation time is sufficiently large enough, the time for creation of threads can be ignored as the probability of affecting the results is minimal.

6.2 Configuring Global and Local Work Sizes

As seen in section 2.3.2, the global and local work sizes are specified when kernel is enqueued in the command queue for execution. The global work size is the total number of threads or work-items that are executed and the local work size is the number of threads that are in a single work-group. Configuring these two parameters properly for each architecture plays a significant role in improving performance. Please observe that the terms “global work size” and “number of threads”, and the terms “local work size” and “work-group size” are used interchangeably in the following sections. In the

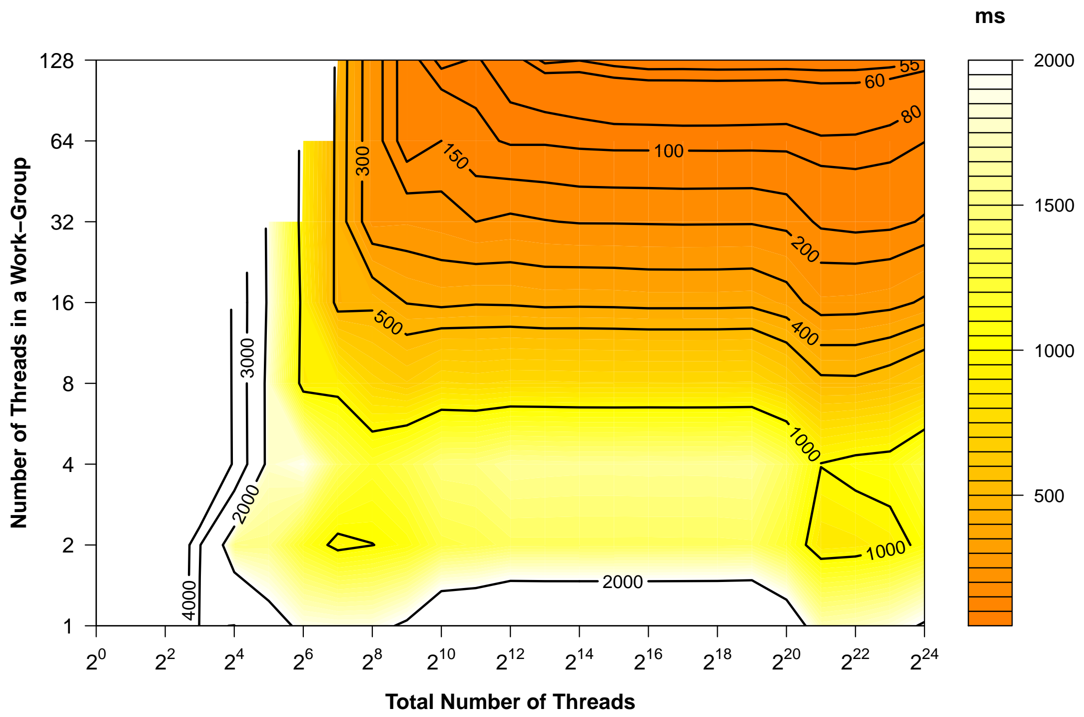


Figure 6.2: ATI

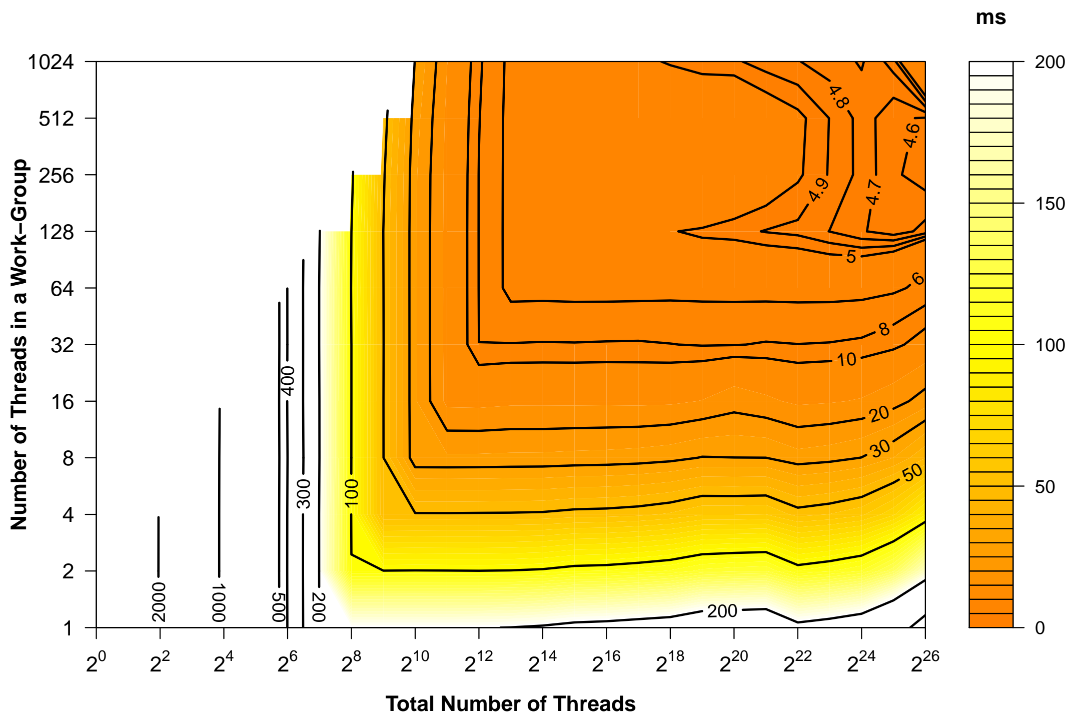


Figure 6.3: Nvidia

next set of graphs, the results of configuring these parameters for the vector-add kernel are explained.

6.2.1 On the GPU

The graphs have been plotted as heat-maps in order to easily identify the combinations of global and local work sizes that are relevant. This perspective helps in giving an overall picture of the optimization space for these two parameters. Figure 6.2 shows the graph for the ATI. The vector-add kernel takes two vectors as input and the size of the vectors are kept constant for a device. The input vector size is determined from the memory capacity of the device. The *CL_DEVICE_GLOBAL_MEM_SIZE* parameter is defined for all OpenCL devices and the total global memory in the device can be retrieved using this parameter.

For e.g., the global memory capacity for the ATI is 256 MB. There are a total of three vectors, i.e two input vectors and one output vector. The elements in the vector are of integer type and so each elements requires four bytes in memory. So, 3 x 4 x the total number of elements in a vector should be less than 256 MB. Evaluating this equation will give the maximum number of elements in a vector to be 2^{23} for the ATI.

In the graph, the x-axis denotes the global work size and the y-axis, the local work size. The ranges for the axes have been determined as follows

- **Maximum Global Work Size -** Each thread processes some part of the vectors and so, the maximum global work size have been determined such that each thread processes at least one element from each vector. So for a vector size of 2^{23}, the number of threads will also be 2^{23}.

- **Maximum Local Work Size -** For determining the maximum local work size, the parameter *CL_DEVICE_MAX_WORK_GROUP_SIZE* is checked, which specifies the maximum number of work-items that can be assigned to a work-group for an OpenCL device.

As can been seen from figure 6.2, the maximum local work size supported on the ATI is 128. For the Nvidia and the Intel devices, the maximum local work size supported is 1024 as shown in figures 6.3, 6.4, and 6.5 respectively.

The graphs have been created by plotting the execution time for every possible combination of local and global work sizes on each device. It can be seen how similar the graphs for the Nvidia and the ATI are, even though they have different architectures.

For both the GPUs, the best performance is seen after the number of threads is more than 4096. The performance is also linearly proportional to the work-group size as execution time keeps decreasing as the work-group sizes are increasing. As per the graphs, the best performance can be obtained from both the GPUs, by having a local work size of 128 and a global work size which is greater than 4096.

On the Nvidia, the performance improves further as the global work sizes are increased more, but the improvement is minimal, with only around 1.2 ms. The best performance at large global work sizes can be attributed to the fact that GPUs have a very large number of computational units and a large number of threads are required to keep them occupied. For the vector-add kernel, the work-grpup size also has to be at least 128 for optimal performance, because with any number lower than that, there are not enough threads to take advantage of the hardware parallelism.

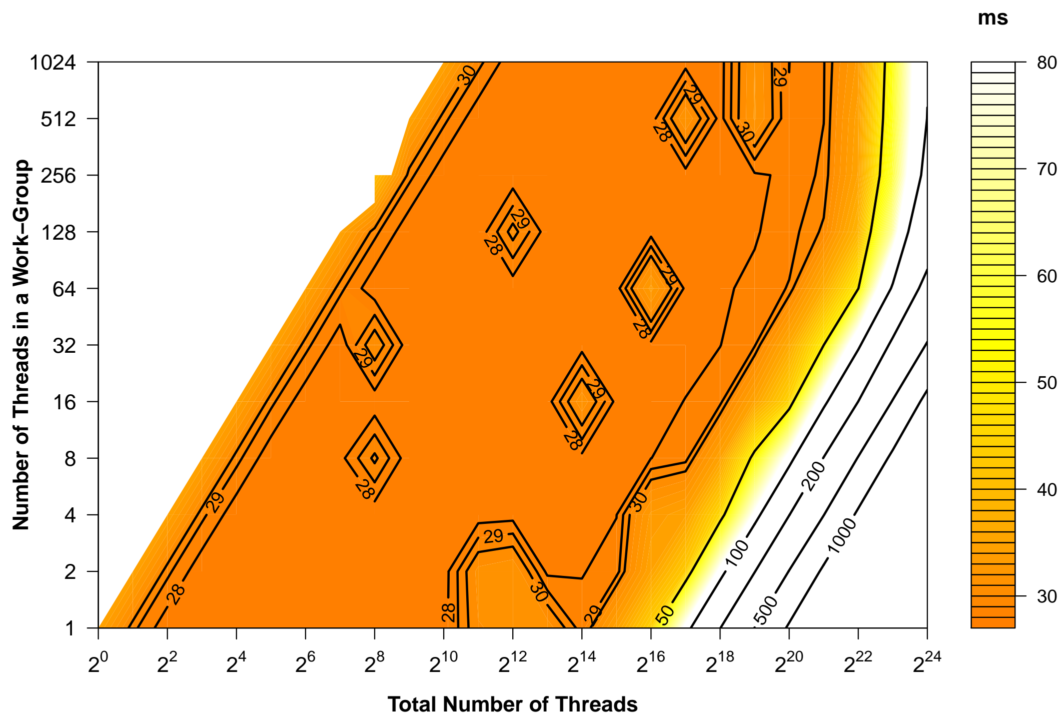


Figure 6.4: Intel i3

6.2.2 On the CPU

On both the CPUs, as seen from figures 6.4 and 6.5, the best performance is seen almost throughout the global work size range. The best execution time is when the number of threads is at least equal to the number of cores in the CPU. So for the i7,

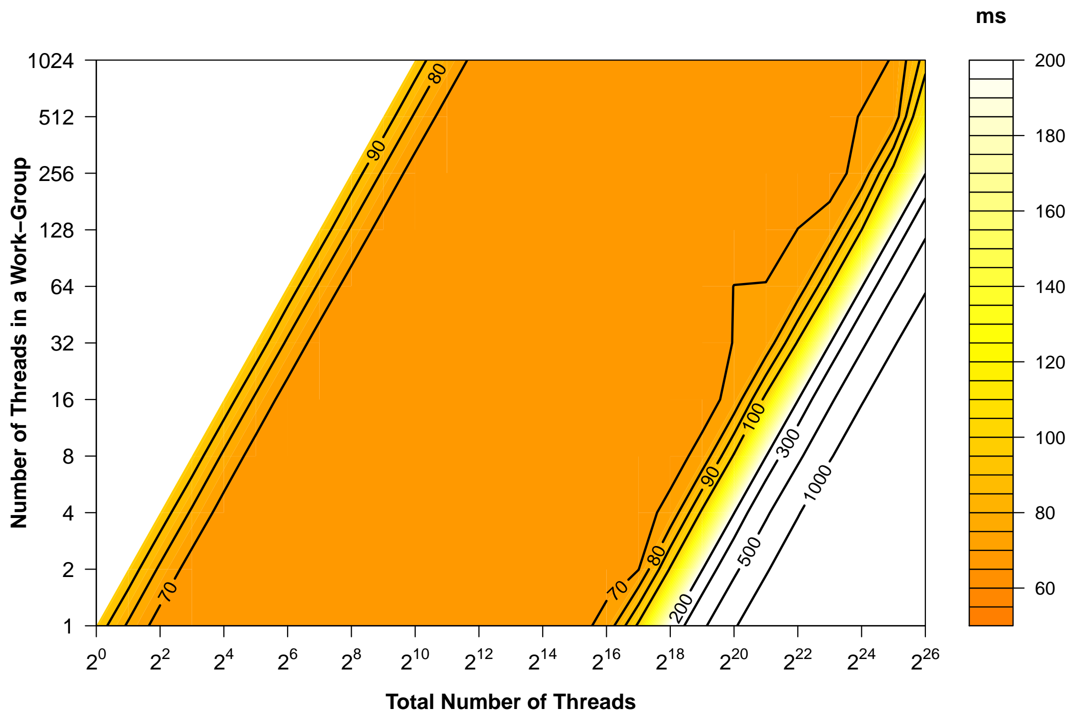
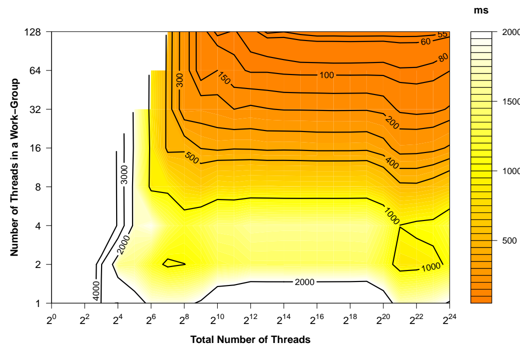


Figure 6.5: Intel i7

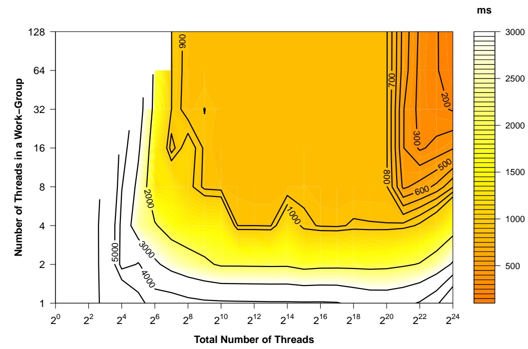
the best execution time starts when the number of threads is equal to six. The optimal local work size increases linearly with the global work size. With a higher number of threads, a larger work-group size is required to keep the best execution time. With more threads scheduled for execution on a core, the data could be stored in the L2 cache for the core. Furthermore, since there are as many threads as the number of elements, almost all the data that is being accessed by the threads could be cached.

6.3 Examining Memory Access Methods

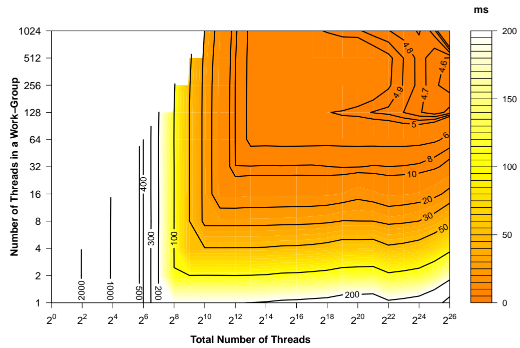
The next optimization explored is the type of memory access method. Memory accesses can be *coalesced* or *non-coalesced*. Two different versions of the vector-add kernel as explained in section 4.3.2. Figure 6.6 shows the graphs for the coalesced and non-coalesced access methods. The graphs which have been presented in the previous section are again presented here along with four new graphs to easily compare both memory access methods. They are plotted as *heat-maps* for identifying the optimal execution times with ease, as the *darker regions* show the best execution times.



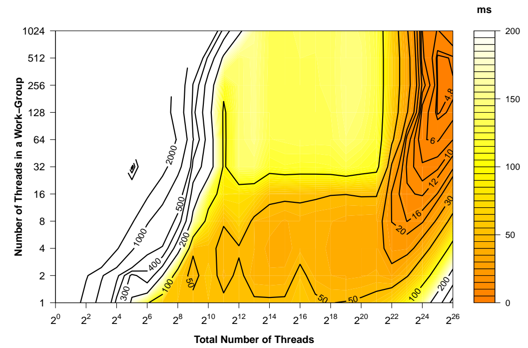
(a) ATI Coalesced



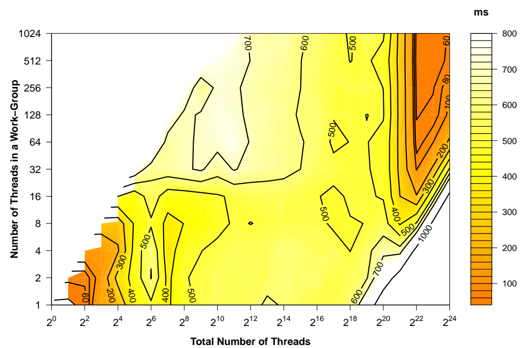
(b) ATI Non-Coalesced



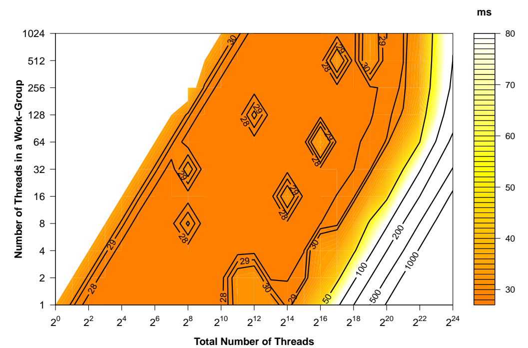
(c) Nvidia Coalesced



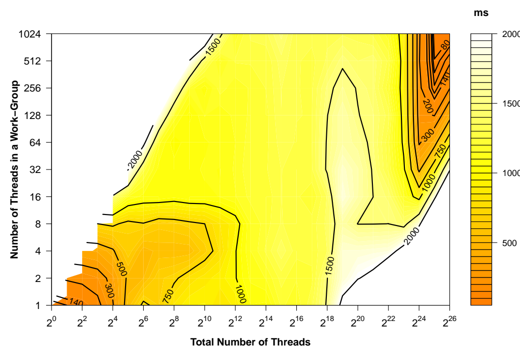
(d) Nvidia Non-Coalesced



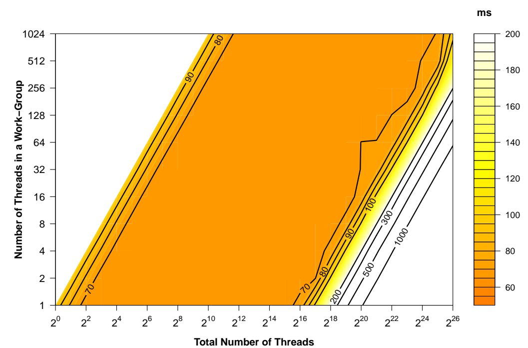
(e) Intel i3 Coalesced



(f) Intel i3 Non-Coalesced



(g) Intel i7 Coalesced

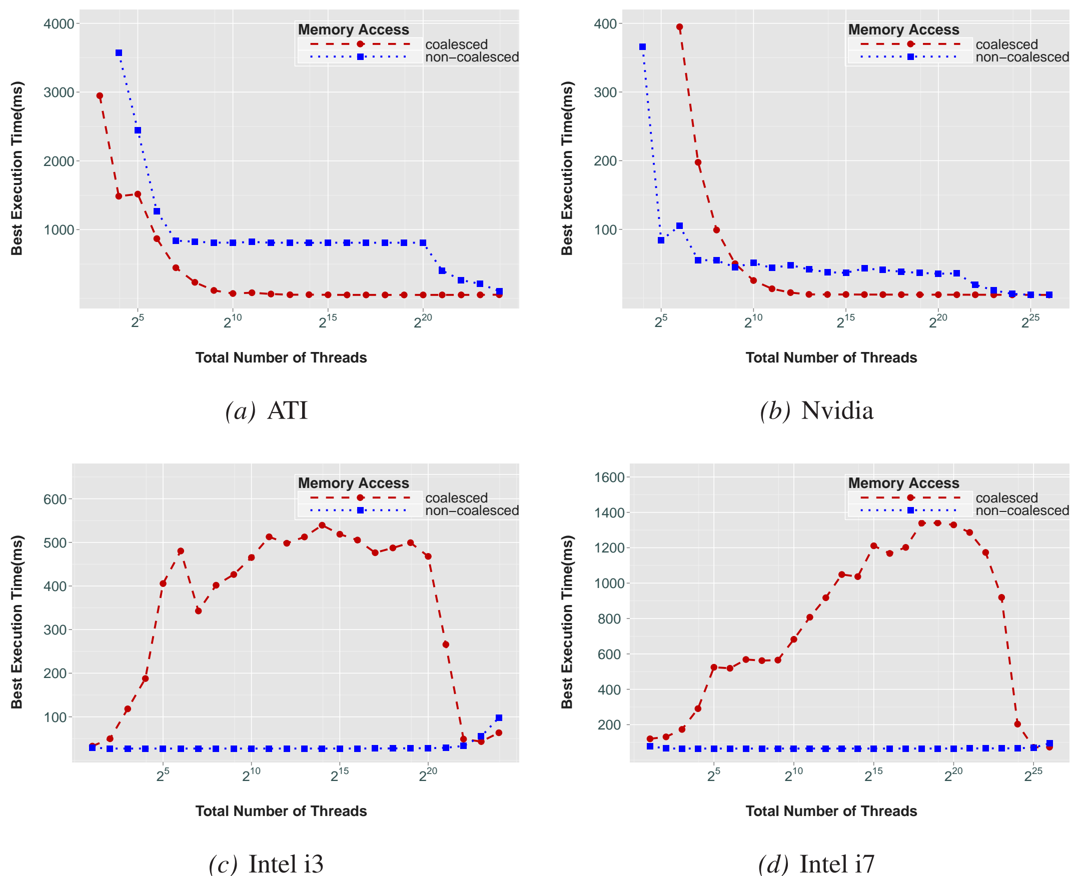


(h) Intel i7 Non-Coalesced

Figure 6.6: The darker regions of the heat-maps show the best execution times. The coalesced versions perform optimally on the GPUs with best performance for large number of threads and work-group sizes. On the other hand, the non-coalesced versions perform best on the CPUs.

6.3.1 On the GPU

In the previous graphs, all ranges of work-group sizes are shown for the whole global work size range. In the next sections, only the optimal local work sizes for the graphs are considered. Figures 6.7(a) and 6.7(b) show that the ATI and Nvidia devices consistently show better performance when the memory accesses are coalesced. With lower number of threads, the choice of memory access method doesn't matter for the GPUs as the performance is equally poor for both methods. As the number of threads rises above 512, it can be seen that there is a significant difference in performance.



(a) ATI

(b) Nvidia

(c) Intel i3

(d) Intel i7

Figure 6.7: Coalesced and non-coalesced access methods on the four devices. The GPUs show optimal performance with coalesced access and the CPUs with non-coalesced access. Regardless of the choice of memory access, equally good performance is seen for all the devices with the highest number of threads.

Coalesced memory access is better for GPUs because of their smaller cache sizes. Since the accesses from the threads in a work-group are for adjacent elements, the data could be cached. Nevertheless, in the case of non-coalesced access, the data accessed is

distributed across the vectors and so even if the GPU caches some data, the next thread access will result in a cache-miss. This means that the bandwidth is being wasted as the processor keeps loading data which is going to be unused into the cache.

Another interesting observation is that, at the highest number of threads, i.e., when *each thread is processing only one element*, the choice of memory access method does not matter. The execution times converge to become similar for both coalesced and non-coalesced accesses. This behaviour can be seen for both the Nvidia and the ATI. Since each thread is accessing only a single element, even for non-coalesced accesses, the data being requested is from adjacent elements.

6.3.2 On the CPU

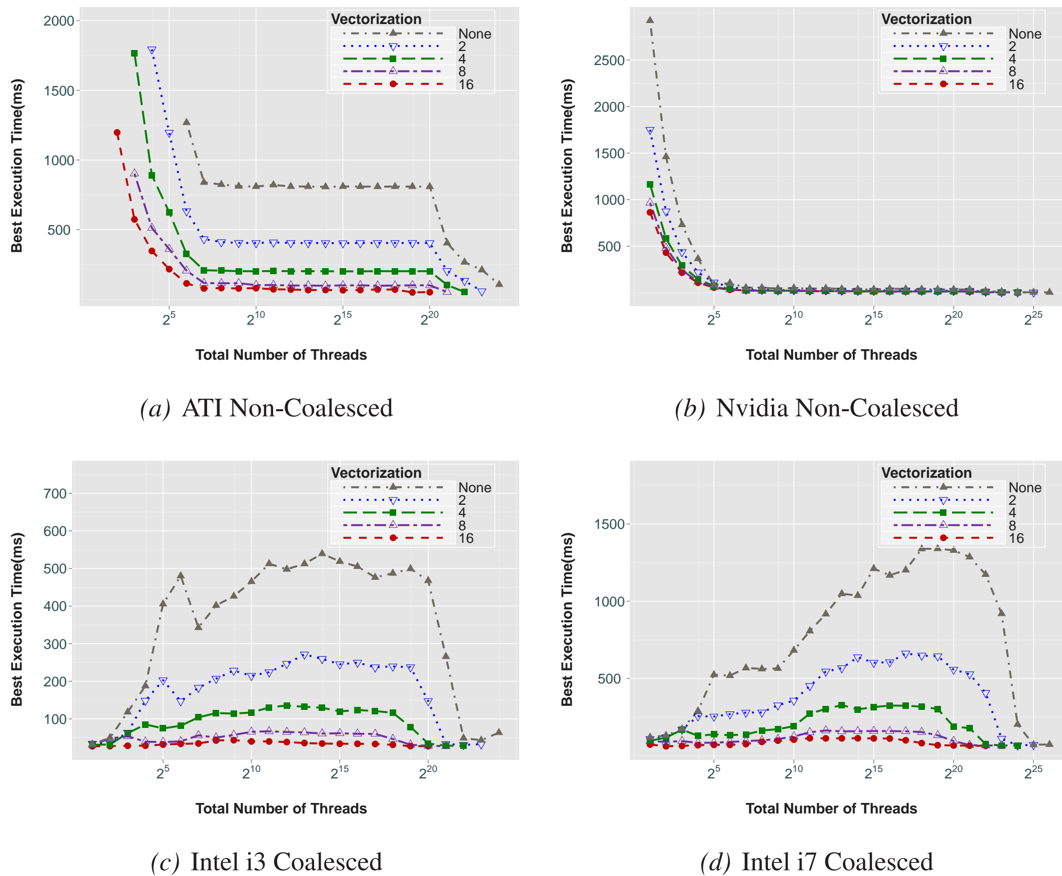
On the other hand, both the CPUs behave differently. Figures 6.7(c) and 6.7(d) show the results for the i3 and the i7 respectively. The CPUs show very good performance with non-coalesced memory access almost regardless of the global work size. Only with very large number of global work sizes, there is a dip in performance. With coalesced access, CPUs show good performance only at extreme ends of the global work size. For 1-2 threads, the performance is somewhat similar to non-coalesced access.

Similar to as with the GPUs, the execution times for both methods converge at the higher end of the global work sizes, independent of the type of memory access. With the larger caches in CPUs, they can keep enough data for the threads in the cache, even if the data is distributed across the input vectors. Thereby the non-coalesced access method is suitable for optimum performance on CPUs.

6.4 Vectorizing the Kernels

In this section, vectorization is explored along with the other parameters mentioned before. Both variants of the vector-add kernel have been modified to support vectorization as mentioned in section 4.3.3. Implementing vectorization enables the OpenCL device to do computation on more than one element (vectors) in a single clock cycle, if vectorization is supported by the device. Experiments are run with four different vectorization factors, *2, 4, 8* and *16*. These factors respectively correspond to *int2, int4, int8* and *int16* as data types for the vector elements in the vector-add kernel.

Firstly, the performance impact of vectorizing the kernel and its implications on the



(a) ATI Non-Coalesced

(b) Nvidia Non-Coalesced

(c) Intel i3 Coalesced

(d) Intel i7 Coalesced

Figure 6.8: Effects of vectorization on the four devices. The Nvidia does not show an improvement in performance whereas the ATI, Intel i3 and Intel i7 show decreasing execution time for increasing vectorization factors.

previous optimizations is explored. For all the four platforms, the worst performing configurations has been chosen, which are the non-coalesced memory access on the GPUs and the coalesced memory access on the CPUs.

6.4.1 On the GPU

Figure 6.8(a) shows the ATI device using a non-coalesced memory access method. As seen before, GPUs had performed poorly with the non-coalesced memory access with no vectorization. However, after vectorizing the code, there is a significant improvement in the performance. The execution time decreases from around 900 ms for no vectorization to around 200 ms for the maximum vectorization factor (16). This can be clearly seen on the ATI, but on the Nvidia (see figure 6.8(b)), there is almost no improvement in performance. This is regardless of the vectorization factor. The ATI

has a vector architecture whereas the Nvidia has a scalar architecture. So the vector data types can be directly mapped to vector instructions which are supported by the ATI, thereby enabling it to process more data per clock cycle. The other graphs for vectorization on the GPUs are included in the appendix B.1.

6.4.2 On the CPU

Just like the ATI, both the CPUs also show a significant improvement in performance with vectorization. Figures 6.8(c) and 6.8(d) show the effect of vectorization on the Intel i3 and the i7, respectively. The graphs shown here are the ones for the coalesced access. The performance improvement is clearly visible for all vectorization factors. The best performance is with the highest vectorization factor for both the CPUs. On the i3, the execution time stays around 30 ms with vectorization. With no vectorization, at the worst case, the execution time increases to as high as 570 ms. This gives a performance improvement of over 91%.

Similarly, on the i7, the execution time has reduced from as high as 1300 ms with no vectorization to a consistent low value of 150 ms, which is again an improvement of over 88%. On both the Intel CPUs, the high performance is due to the SSE instructions generated by the compiler. With SSE instructions, scalar instructions can be packed together to be executed more quickly. To see the other graphs with vectorization for the CPUs, please refer to the appendix B.1.

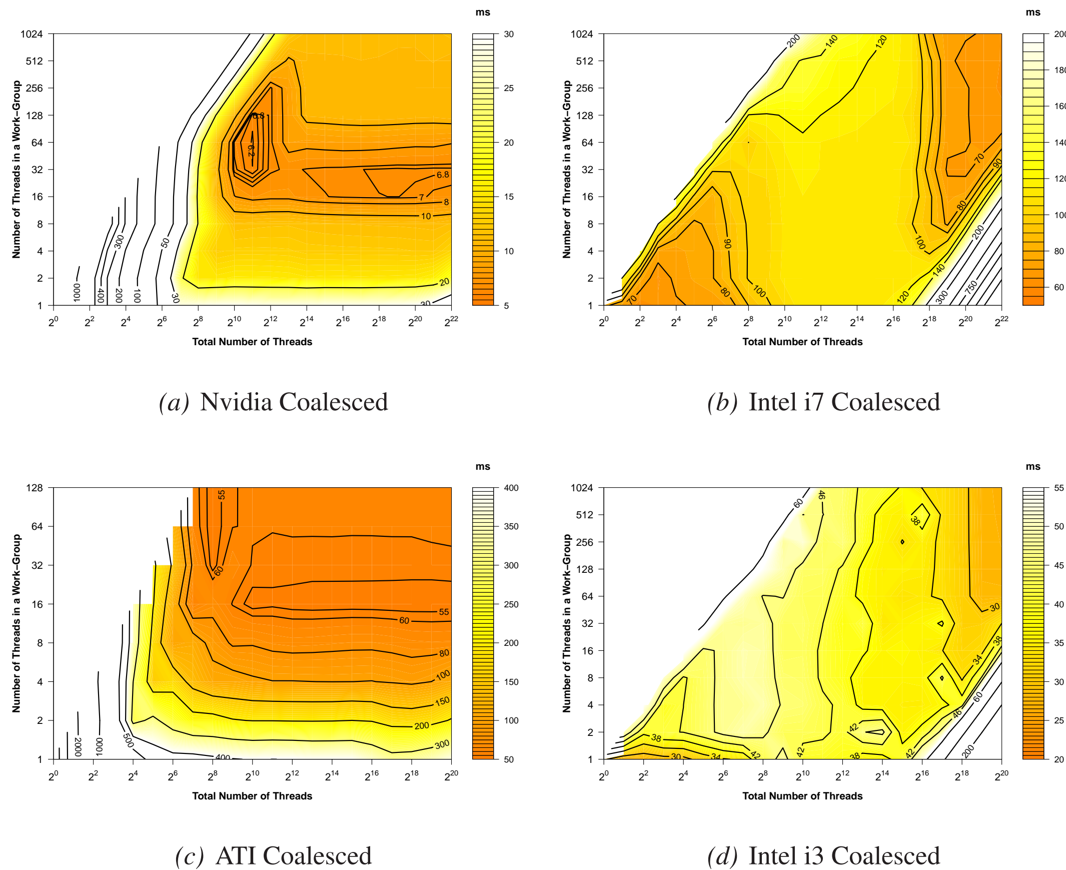
6.4.3 Global and Local Work Sizes with Vectorization

After vectorizing the kernel, there are some expected and some unexpected observations which are seen from the results. The heat-maps for the devices with a vectorization factor of 16 are shown in figure 6.9. The rest of the graphs are provided in the appendix B.2.

6.4.3.1 On the GPU

On the Nvidia, shown in figure 6.9(a), there is a small region available on the contour map which shows the least execution time. Bear in mind that the kernel with coalesced access is shown here, which have had good performance before vectorization.

Before vectorizing the kernel, with global work size greater than 1024 and local work size above 128, the Nvidia could get good performance. Though unexpectedly,



(a) Nvidia Coalesced

(b) Intel i7 Coalesced

(c) ATI Coalesced

(d) Intel i3 Coalesced

Figure 6.9: Effects of vectorization on choosing global and local work sizes. The Nvidia shows a decrease in performance with vectorization, whereas the ATI shows a decrease in execution time. Previously on the CPUs, the optimal regions were closely confined to the extreme ends and with vectorization, this has improved remarkably with drastic reduction in execution times.

the best execution time have actually increased from 4.6 ms to 6.2 ms, and the relatively large area of best performance has been reduced to the area where the local work size is between 32 and 128 and the global work size is between 256 and 2048. This shows that for the Nvidia architecture, vectorizing the kernel can be detrimental to performance. With the ATI, as can be seen in figure 6.9(c), there is no difference in performance with the best execution times being the same as before for coalesced access. The non-coalesced access also shows a similar graph confirming the previous evaluation that vectorization is a good optimization for the ATI architecture.

6.4.3.2 On the CPU

Figure 6.9(b) shows the effect of vectorization for the i7. The version shown here is the coalesced one. The graphs for the non-coalesced version and the ones for the i3 exhibit

the same behaviour. They are included in the appendix B.2 for reference. Previously, the good performance is seen at the extreme ends for both the CPUs and that was with the coalesced access. However, after vectorizing the kernel, good performance can be seen throughout the range of global work size. The best performance is still seen at the extreme ends, but even if those values are not chosen, a relatively good performance is seen on the CPUs.

6.4.4 Memory Access Methods with Vectorization

The effect of vectorization on the memory access choice is now analysed. The graphs are plotted considering the minimum execution time across different vectorization factors, considering only the optimal local work sizes.

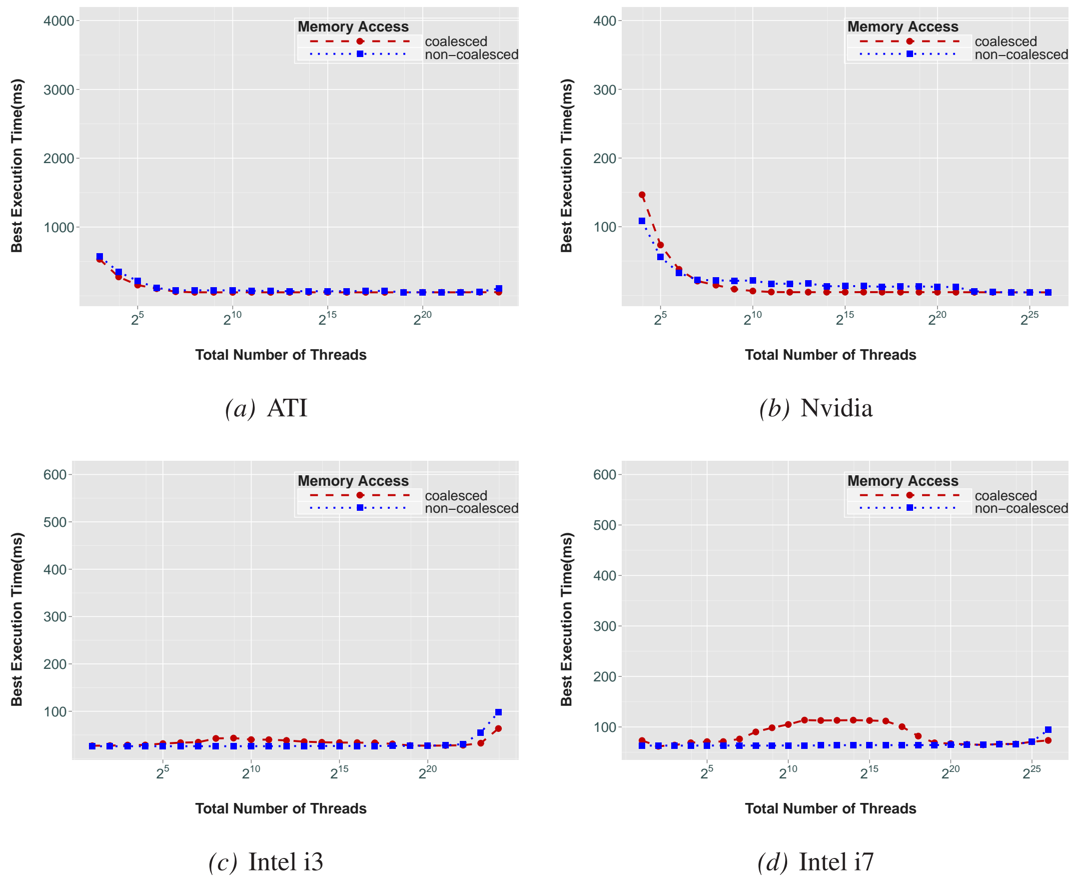
6.4.4.1 On the GPU

Figure 6.10(a) shows impact of vectorization for the coalesced and non-coalesced memory accesses with the ATI. It can be seen that the execution time for non-coalesced access has drastically reduced and is almost the same as that for the coalesced access.

With vectorization, more elements are accessed and computed in fewer clock cycles. So the number of cache-misses will be much lesser than before. For e.g., with a vectorization factor of int16, 16 elements are processed at the same time. So even if the data is cached by the GPU, the probability of the next access being a cache-hit is high, and hence the high performance on the ATI. Figure 6.10(b) shows the results for the Nvidia and the difference which is seen here is also as expected. Vectorization does provide an improvement on Nvidia but not as much as the ATI. The non-coalesced access is still slower than the coalesced access for almost throughout the global work size range.

6.4.4.2 On the CPU

On the CPUs also, like the ATI, there is a significant improvement in performance. Figures 6.10(c) and 6.10(d) show the results for the i3 and the i7 respectively. The primary focus here is the variance in execution time. On the Nvidia the improvement in performance was not that significant. However, both the i3 and the i7 show a drastic reduction in execution time with vectorization. In the middle ranges for the global work size, the non-coalesced access is still faster. This can be due to the thread switching and caching overhead not being overcome by the benefit with vector instructions. An



(a) ATI *(b)* Nvidia

(c) Intel i3 *(d)* Intel i7

Figure 6.10: Effects of vectorization on the choice of memory access method. With vectorization, both the memory access methods become almost equally good for the ATI and the CPUs. There is a huge improvement in performance on these devices relative to the performance without vectorization. The Nvidia has no impact on performance.

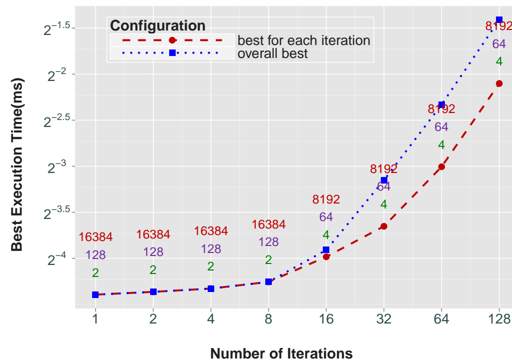
overall observation is that vectorization will certainly improve performance on the CPUs.

6.5 Evaluating the "Compute-Adaptable" Kernel

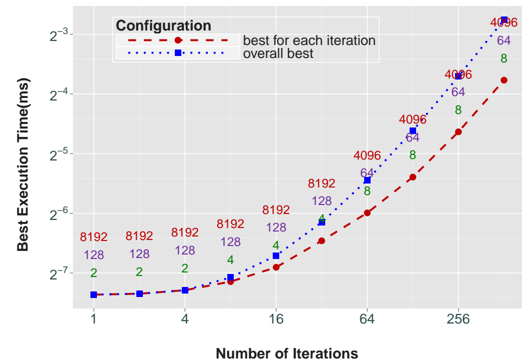
In evaluating the compute-adaptable kernel, the goal was to find the best configuration for a certain level of computation. The figure 6.11 show the results for ATI, Nvidia, i3 and i7 respectively.

The x-axis denotes the number of iterations for the extra loop that was added in the kernel. The y-axis as before denotes the best execution time, but is plotted in log scale. The graphs are plotted considering the best execution time for each number of iterations. For each point plotted, the configuration of the system for that particular

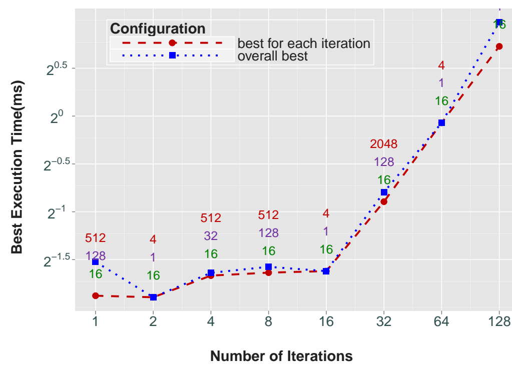
execution time has been noted. Three numbers are noted for each point which is from top to bottom, the global work size, the local work size and the vectorization factor respectively.



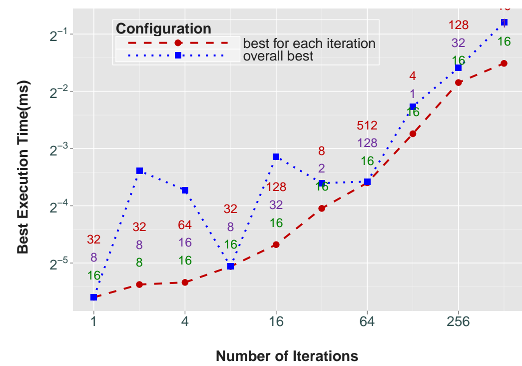
(a) ATI Coalesced



(b) Nvidia Coalesced



(c) Intel i3 Coalesced



(d) Intel i7 Coalesced

Figure 6.11: The optimal configurations for each iteration factor on the compute-adaptable kernel. Both the GPUs show consistent behaviour with larger number of threads and lower vectorization when the kernel is memory-bound and vice-versa when compute-bound. However, both CPUs show random optimal configurations with no observable pattern. The best vectorization factor seems to be 16 for the CPUs as it appears most frequently.

6.5.1 On the GPU

For figure 6.11(a), which is the ATI, the kernel begins to be compute-bound after the number of iterations are larger than 16. An interesting observation is that the *number of threads* needed for better performance is *double* the number needed after the kernel begins to be *compute-bound*. This can be due to the fact that at lower number of iterations, the work done by a single thread is lower and so more threads are needed

to keep the processing elements occupied. The local work size is also halved when the number of iterations crosses 16, whereas the vectorization factor doubles. However, these can be explained as they are dependant on the global work size.

The local work size has to be doubled at the smaller number of iterations, because with the larger global work size, the execution time will be higher if lesser number of threads are allocated to a work-group. The reason for the vectorization factor being higher when the kernel becomes compute-bound could be that there are lower number of threads than before and more computation also and so a higher vectorization factor will be appropriate for lower execution time.

Then the question arises as to why the vectorization factor is not higher. The reason could be that, with an even higher vectorization factor, the *benefit gained* might be overruled by the *higher memory latency*. Figure 6.11(b) shows the graph for the Nvidia. The only difference here is that the kernel becomes compute-bound when the number of iterations become larger than 64. The doubling of the vectorization factor and the halving of both the global and local work sizes are similar to the behaviour for the ATI.

6.5.2 On the CPU

Figures 6.11(c) and 6.11(d) shows the behaviour for the CPUs. Though the graphs for the GPUs have been very consistent with expectations, the graphs for CPUs appear to exhibit more random behaviour. For the i3, there are some abrupt jumps and the configurations are also quite random. For the i7 also, the configurations seem to be arbitrary. Even though the kernel starts to be compute-bound for the i7 when the number of iterations crosses 16, there is no observable pattern of configurations.

Nevertheless, there are some interesting observations, one of which is that, the number of threads is much lower than that for the GPUs. The highest value for the i3 is 2048 and that for the i7 is 512. Another interesting point to note is that at almost all iteration levels, the *best vectorization factor* seems to be 16. The graphs for the coalesced and non-coalesced versions exhibit similar behaviour and hence the results for the coalesced versions are shown here. For the non-coalesced results, please refer to the appendix B.3.

6.6 Summary

This chapter presented the experiment results and the critical analysis of the results. The results for all the various kernels presented in Chapter 4 are analysed for the four devices. For each optimization parameter the optimal configuration was presented. The effects of one parameter on others are also discussed. When designing applications for the GPU, using coalesced access wherever possible will be optimal. For implementing this, the source code for the kernel will have to be modified. However, as the results show, in case the memory access choice cannot be made, the application could be configured to use a large number of threads resulting in better performance. In case of the CPU, the non-coalesced access method is always optimal. Nevertheless, vectorizing the kernel will be a good method to get optimal performance on the CPU even if the kernels are implemented with coalesced memory access. Another area of discussion was the effects of exploring the same optimization parameters on memory-bound and compute-bound kernels.

The chapter also compared and summarized the performance of various device architectures for each of the experiments. The next chapter presents the difficulties encountered, the future work and finally, the conclusion to the thesis.

Chapter 7

Conclusion

This thesis has explored the optimization space of multi-core architectures such as ATI, Nvidia and Intel using various OpenCL benchmarks. The benchmarks are custom developed to be configurable for applying exhaustive combinations of optimization parameters, for the exploration process. The design and implementation for the kernels have been provided in chapter 4, with discussions about the optimization space and the kernel variants designed for evaluating each optimization. Finally in chapter 6, the results of the experiments have been provided, including the optimal configurations for each architecture for each optimization parameter.

This chapter first summarizes the contributions of the thesis. In the subsequent section, the difficulties encountered during the thesis are presented, followed by a discussion of the future work. Finally, section 7.4 provides a summary of this work, thereby concluding the thesis.

7.1 Contributions

The aim of this project has been to explore the optimization space using OpenCL benchmarks and identify the optimal parameters for multi-core architectures. The experiment results and analysis for the architectures are summarised below.

Choosing the number of threads The parameters for the number of threads are configured when the kernel is enqueued for execution. Two parameters are configured, which are the total number of threads (global work size) and the number of threads in a work-group (work-group size). On the GPUs, the best performance is always visible when the *total number of threads* are much *higher* (< 4096)

than the *number of processors*, regardless of other optimizations. Another point to note is that the performance increase is linearly proportional to both the total number of threads and the number of threads within a work-group. So *increasing the work-group size* as high as possible also results in good performance on the GPUs. This behaviour can be attributed to the fact that GPUs have a large number of computational units, thereby requiring a large number of threads to take advantage of hardware parallelism.

On the CPUs, choosing the number of threads depends on the memory access method implemented. With the non-coalesced access, choosing the number of threads for the best performance is quite easy as the only condition is that the *number of threads* has to be at least *equal to the number of cores*. So for the Intel i7, optimal performance is with 6 or more threads. However, with coalesced access, the number of threads has to be either equal to the number of cores and or as high as possible to derive the best performance. Obtaining *good* performance with a *very large* number of threads is an interesting behaviour on the CPUs. This burst in performance could be due to the *less amount of context switching* as the amount of computation done by one thread is minimal. There are as many threads as the number of elements and so, almost all the data being accessed by a thread could be stored in the large caches of the CPU.

Choice of memory access method *Coalesced* memory access is the best choice for *GPUs* whereas *non-coalesced* access is the optimal one for *CPUs*. However, there are certain cases where the non-coalesced access can be optimal for GPUs and coalesced access good for CPUs. Coalesced access is better for GPUs due to their *small cache sizes*. With non-coalesced access, the bandwidth is wasted resulting in bad performance, as the processor keeps loading data which is going to be unused into the cache.

If the *number of threads* is as large as the *data* such that the computation being done by one thread is minimal, then the choice of memory access *does not* matter. This is because in this case there is less unused data in the cache. Similarly, the other access method could be opted on architectures where *vectorization* has a *profound effect*, namely the ATI and the Intel. After vectorization, the performance for both the access methods is optimal for these architectures.

Vectorizing the code Vectorization is an important optimization in the sense that on architectures that support it natively there are huge improvements. Vectorization

as an optimization parameter have been investigated by Rul et al.[43] also. The improvements gained in some cases are large enough to overcome the negative impact of other parameters. Nvidia *does not* show much improvement with vectorization and as the experiment results show, in some cases it is even detrimental to performance. On the other hand, both the *ATI* and the *Intel* architectures *benefit heavily from vectorization* as seen from the results. ATI supports vector instructions natively, thereby enabling it to process more data per clock cycle. On the ATI, the non-coalesced memory access which initially had low performance exhibits similar performance to coalesced memory access after applying vectorization. With vectorization, more elements are accessed per clock cycle. So the number of cache-misses drastically reduces. Similar is the case for the CPUs as they both react positively to vectorization.

Optimal configuration for memory-bound and compute-bound kernels The experiments have been run with memory-bound and compute-bound kernels for achieving two goals.

- To investigate the *effects of increasing computation* on the aforementioned optimization parameters.
- To identify the *optimal configuration* for a kernel having a certain level of computational intensity.

On the ATI, as the kernel becomes compute-bound, the total *number of threads* for the best performance gets *halved*. The work-group size also gets halved whereas the vectorization factor doubles. The reason for the number of thread being halved can be due to the fact that at a lower *computational intensity*, more threads are needed to keep the *processing elements occupied*. When the kernel becomes compute-bound, the vectorization factor doubles as there are lower threads than before and more computation, thereby requiring higher vectorization factors for lower execution times. Both the GPUs show very similar behaviour whether they are memory-bound or compute-bound. The CPUs on the other hand, exhibit *random behaviour*. There is no identifiable pattern other than that for most cases, the *best* vectorization factor is *16*. So for the CPUs, it is difficult to predict the optimal configuration of parameters.

7.2 Difficulties Encountered

In this section, some of the major difficulties encountered during the project are discussed. During the initial stages of the project, we faced an issue in which the test verification for the computation done by the kernel passes successfully, but the computation takes less time than expected. With further investigation, it was found that the *GPU memory is retaining the data* from the previous execution and even though no computation took place, the *old data* was being copied back and verified. To prevent this problem, the GPU memory was reset each time by copying zeroed data into it before running a kernel.

Another issue was the *faulty memory* of the Nvidia Geforce GTX 580 which was the high-end GPU initially considered. With the exhaustive number of experiments, all the devices have been *thoroughly stressed* to their limits. The experiments used the whole of the global memory in the devices. During the test verification, the test execution program reported that some of the *computations* done by the GTX 580 were *incorrect*. With further investigation, the memory was found to be faulty and the card was replaced by the Nvidia Tesla C2070.

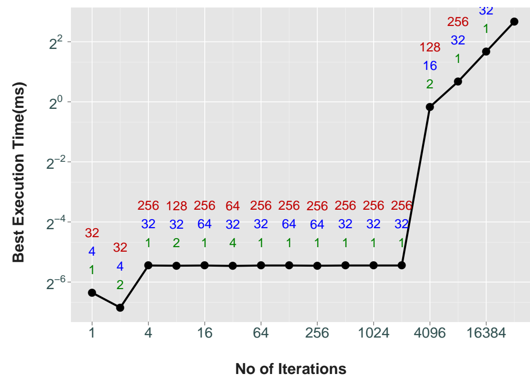


Figure 7.1: ATI with the problem

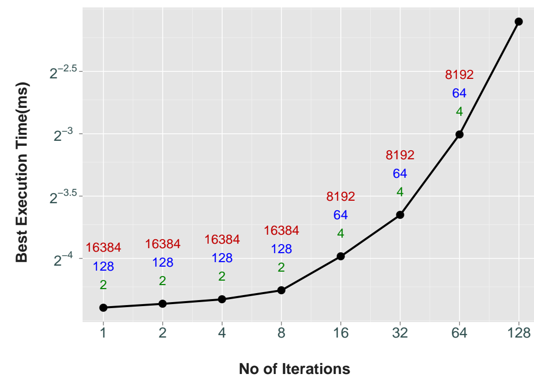


Figure 7.2: ATI

The largest setback occurred while implementing the "compute-adaptable" kernel. The initial design of the kernel just contained the additional loop with the loop-count and loop-increment variables passed as macros into the kernel. It did not contain the additional arguments, as explained in section 4.4. Experiments were run on all architectures with the original version of the compute-adaptable kernel. However, on the ATI, there was a problem of an *abrupt jump* in the obtained results. This problem can be clearly seen in the figure 7.1. Figure 7.2 shows the graph after the workaround was applied. On the other architectures, there was no such issue.

The *instruction set architecture* (ISA) code generated for the ATI and the *PTX* code generated for the Nvidia, were compared to understand this behaviour. The ISA[1] is the assembly code for the AMD, and similarly the PTX[36] is the assembly code for the Nvidia. The ATI compiler identifies the *loop invariant* code in the kernel and optimizes it away, resulting in the low execution time. However, the problem was that this is done only for a certain number of iterations, after which the compiler stops optimizing, causing the abrupt jump in execution time. For a vector size of 256, the optimization stops after the number of iterations become greater than 2048. The optimization is certainly a good job done by the compiler, but the results will be *biased* for the experiments as the computation is *not* actually being done.

For further investigation, the compiler optimizations were disabled using the –*cl_opt_disable* flag. The Nvidia compiler worked as expected and obeyed the flag. Although on the ATI, the result was interesting as the ATI compiler did not respond to this flag as it still did the optimization. This was confirmed by checking the ISA code. So this is hence confirmed to be a *bug* with the ATI compiler[1].

To fix the issue, the kernel had to be redesigned in some manner such that the compiler will not be able to identify loop invariant code. Each time after the kernel was tweaked, the ISA code had to be analysed and compared to and so this process was time-consuming. Initially, the loop-count was passed across to the kernel through a macro, and this was changed to a *dynamic parameter* to prevent the optimization. Once this was done, the loop invariant optimization was prevented, but the compiler began to apply loop unrolling instead. Finally, the kernel was redesigned such that both the loop-count and the loop-increment variables are *passed as arguments* to the kernel function. This version worked as expected and successful results were obtained as shown in figure 7.2.

7.3 Future Work

Within the limited time-frame for the project, various optimization parameters are explored by adapting the vector-add application to create different versions for each. From evaluating the results of all the various kernel versions, the optimization parameters could be mapped as either positive or negative for the evaluated architectures. The results also confirm that an optimization which works well for a particular architecture, can impact negatively on another architecture. However, there is still room for

[1]This bug is fixed in the latest version of the ATI compiler released on August 8th.[53]

improvement in many areas.

One possibility is to *automate the process* of exploring the optimization space, such that the optimal program can be found automatically. A lot of manual work is required to quantify the performance of each parameter and tweaking the kernel subsequently. Prior work is done in the auto-tuning area for CUDA[33, 9, 7], which could be adapted to create an auto-tuning framework for OpenCL also. One trivial method of implementing auto-tuning is to run all possible configurations and then pick the best one. This could be even improved upon by applying heuristics to make the process faster.

In this project, the one-dimensional vector-addition and its variants have been used for the experiments. *More applications* could be added for experimentation, thereby further improving the results. With more time, two-dimensional applications like matrix multiplication or even three-dimensional applications such as 3D FFTs[33] could be evaluated. Another improvement is to evaluate *more architectures* for further exploration of the optimization space. The Cell[16] is one such architecture that could be included in future evaluations.

Another area of improvement could be in *applying* the obtained results. It could be used by developers for *manually tuning* the programs for a particular architecture. A better option is to incorporate the optimization results into a *machine learning model* to be used in a *compiler*. By just specifying the target architecture at compilation, the compiler will then be able to apply the right optimizations for the OpenCL program, yielding maximum performance. This will make the application of optimization to the program, a transparent and a programmer-independent process, similar to the optimization currently done by standard compilers such as GCC[13].

7.4 Summary

In this thesis, the optimization space have been exhaustively explored by running OpenCL benchmarks (over 600,000 runs) for all the possible combination of optimization parameters on the multi-core architectures. The optimal parameters have been successfully identified and in addition, the rationality for the parameters being the best is also sought out. The results could be used by programmers developing applications for these architectures to improve the performance significantly. Furthermore, the parameters can be also used for porting existing applications for these architectures with optimum performance gains. As mentioned in the previous section, the identified pa-

rameters could also be incorporated into a compiler for automatic optimization based on the target architecture.

The experiments also show that each architecture responds differently to various optimizations and hence a thorough exploration of the optimization space is required for maximum efficiency. A limitation of the approach is that the optimization space have been manually explored, which is a very time-consuming process. Another approach that could be adopted is automatic exploration, but the sensitivity of the architectures to the optimization parameters could be a problem.

In this chapter, the contributions of this thesis were presented in the first section. The various optimization parameters along with the right configurations for them for different architectures were presented. The chapter also discussed the difficulties encountered during the thesis. Finally, the thesis is concluded with a discussion of the various areas of future work for this thesis.

Appendix A

Kernel Implementation

A.1 Coalesced Vector-Add

```
__kernel void vector_add(__global VECTOR_TYPE *A,
                         __global VECTOR_TYPE *B,
                         __global VECTOR_TYPE *C)
{
        int i = get_global_id(0);
        int step = get_global_size(0);

        int start = i;
        int end = NO_OF_ELEMENTS / VECTORIZATION;


        for(int j = start; j < end; j = j + step)
                C[j] = A[j] + B[j];
}
```

Listing 5: Coalesced Vector-Add

A.2 Non-Coalesced Vector-Add

```
__kernel void vector_add(__global VECTOR_TYPE *A,
                         __global VECTOR_TYPE *B,
                         __global VECTOR_TYPE *C)
{
        int i = get_global_id(0);
        int no_of_threads = get_global_size(0);

        int elements_per_thread = NO_OF_ELEMENTS / no_of_threads;
        int n = elements_per_thread / VECTORIZATION;

        int start = i * n;
        int end = (i+1) * n;

        for(int j = start; j < end; j = j + 1)
                C[j] = A[j] + B[j];
}
```

Listing 6: Non-Coalesced Vector-Add

A.3 Non-Coalesced Compute-Adaptable Vector-Add

```
__kernel void vector_add(__global VECTOR_TYPE *A,
                         __global VECTOR_TYPE *B,
                         __global VECTOR_TYPE *C,
                         __global int iter,
                         __global int inc)
{
        int i = get_global_id(0);
        int no_of_threads = get_global_size(0);
        int elements_per_thread = NO_OF_ELEMENTS / no_of_threads;
        int n = elements_per_thread / VECTORIZATION;
        int start = i * n;
        int end = (i+1) * n;

        VECTOR_TYPE a,b,val;
        int c = inc;

        for(int j = start; j < end; j = j + 1)
        {
                a = A[j];
                b = B[j];
                val = a + b;
                int k=0;
                for(; k< iter;)
                {
                        val = val + k; k = k + c;
                }
                C[j] = val;
        }

}
```

Listing 7: Non-Coalesced Compute-Adaptable Vector-Add

Appendix B

Experiment Results

B.1 Vectorization - Memory Access Methods

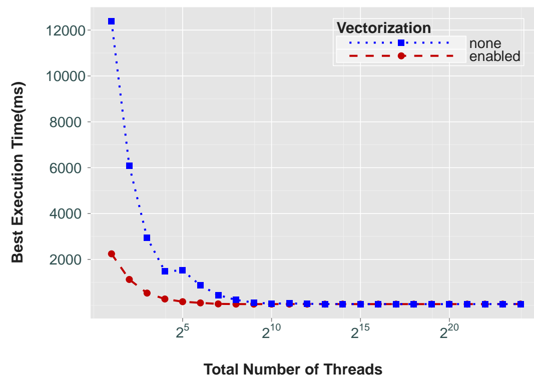


Figure B.1: ATI Coalesced

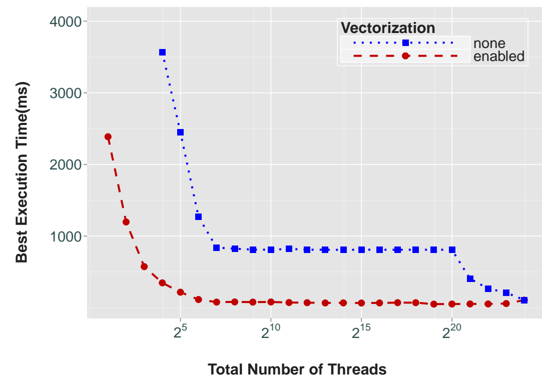


Figure B.2: ATI Non-Coalesced

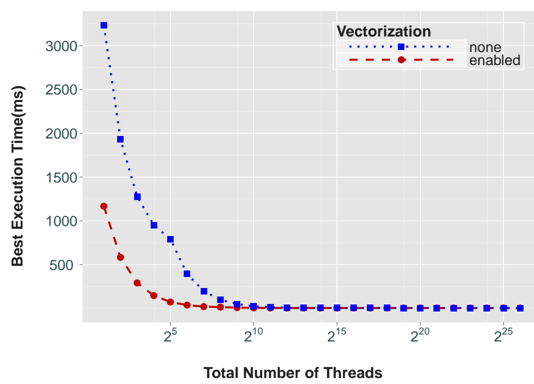


Figure B.3: Nvidia Coalesced

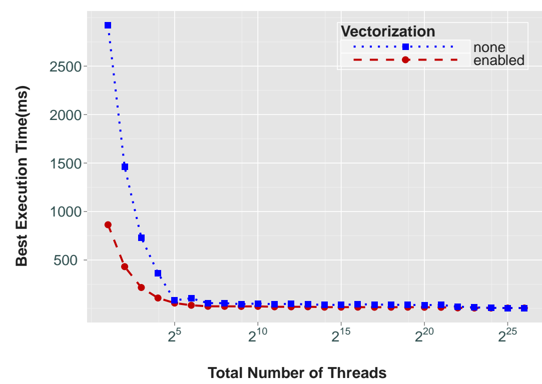


Figure B.4: Nvidia Non-Coalesced

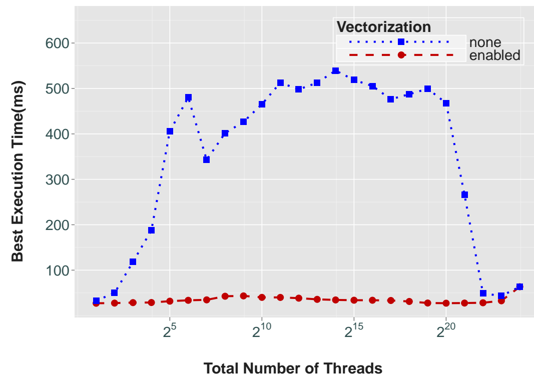


Figure B.5: Intel i3 Coalesced

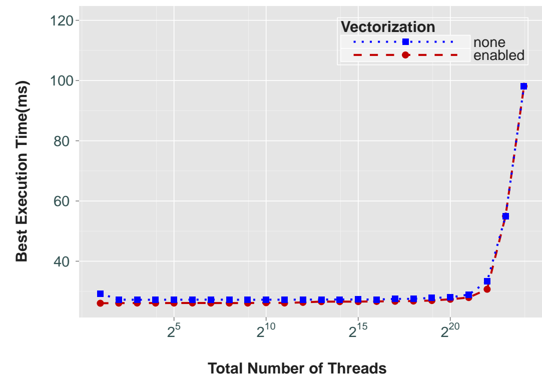


Figure B.6: Intel i3 Non-Coalesced

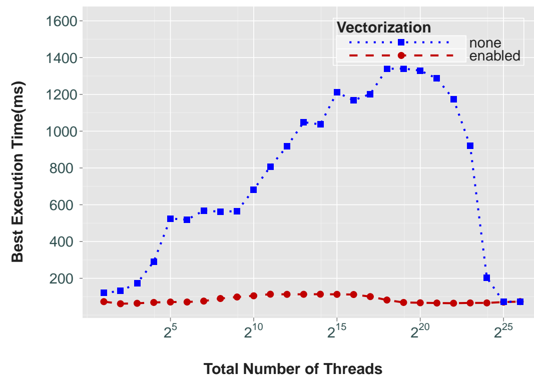


Figure B.7: Intel i7 Coalesced

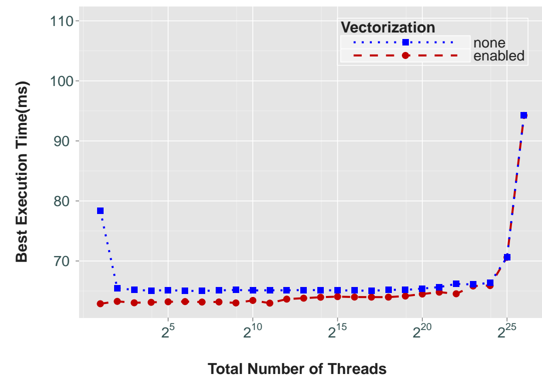


Figure B.8: Intel i7 Non-Coalesced

B.2 Vectorization - Global and Local Work Sizes

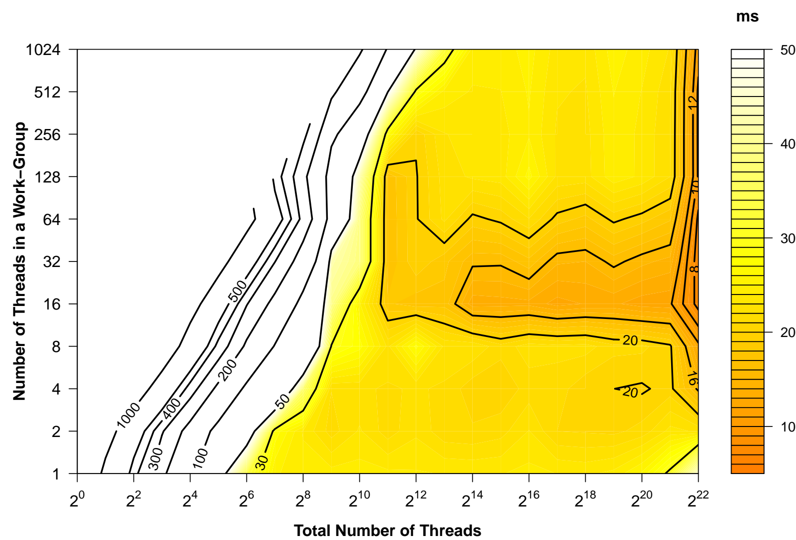


Figure B.9: Nvidia Non-Coalesced

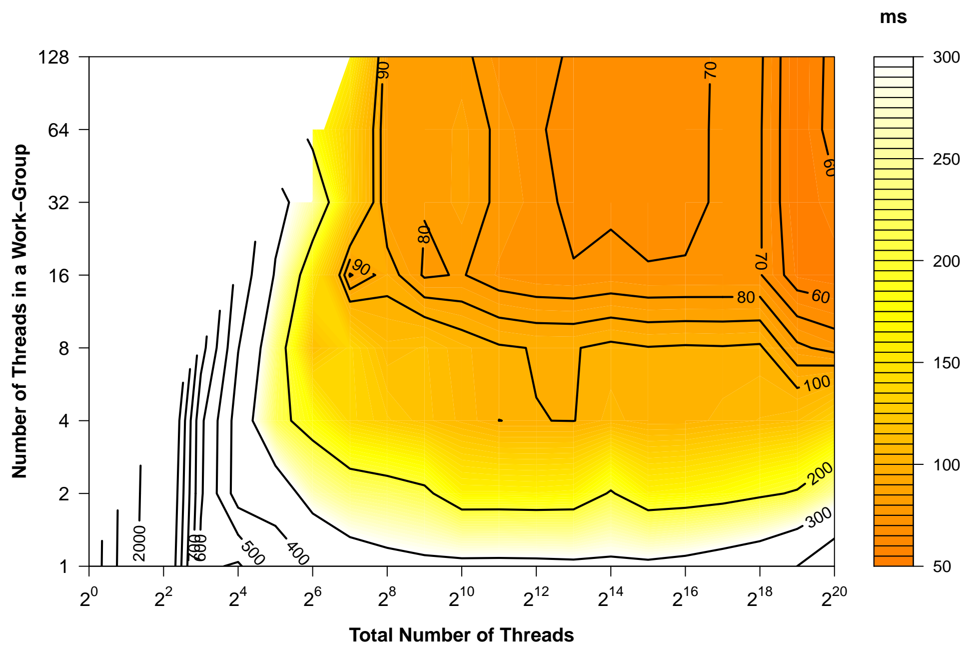


Figure B.10: ATI Non-Coalesced

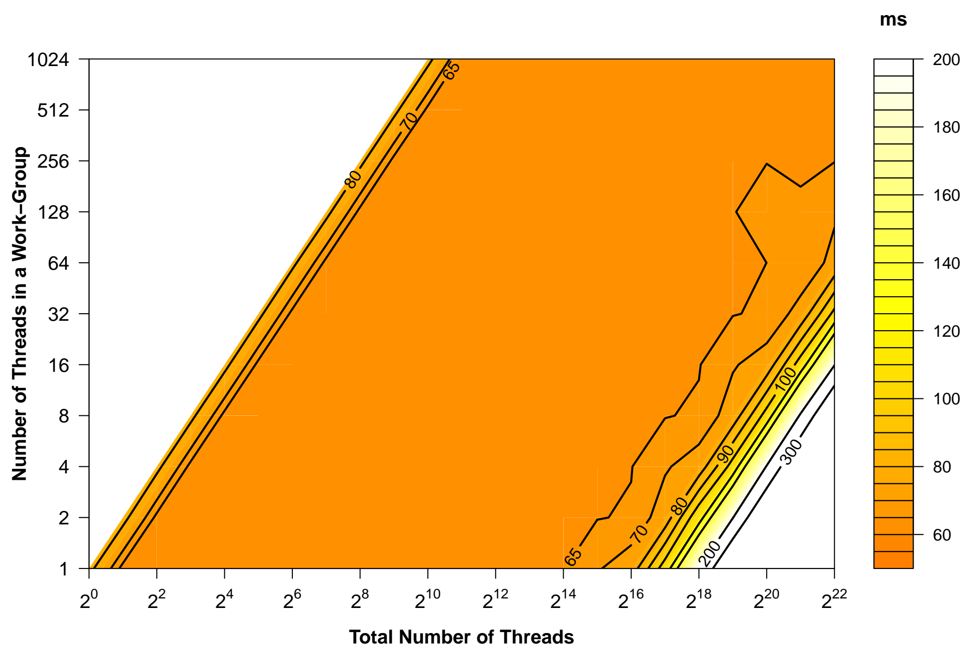


Figure B.11: Intel i7 Non-Coalesced

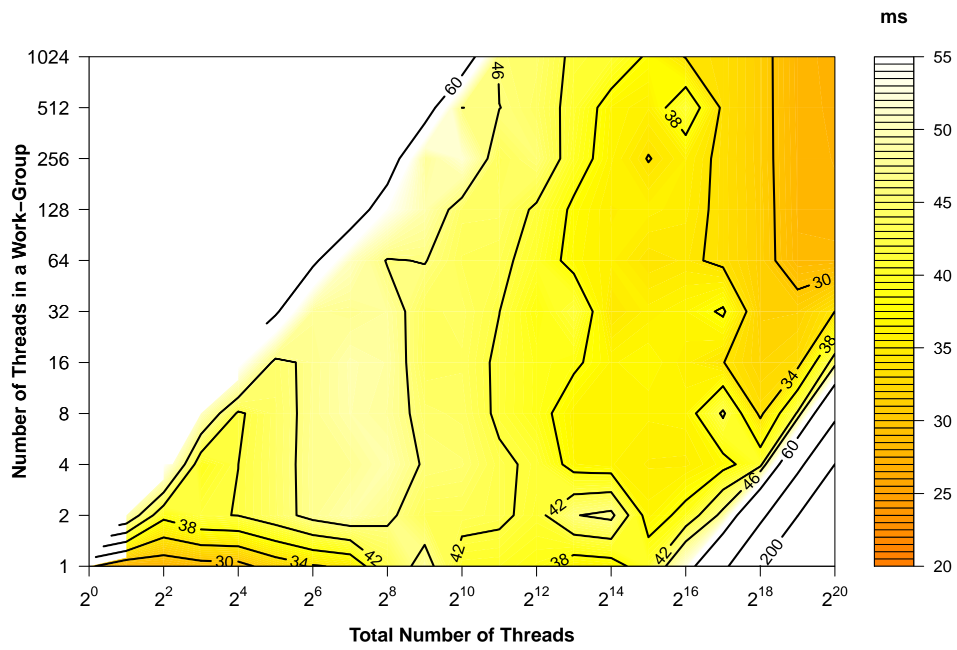


Figure B.12: Intel i3 Coalesced

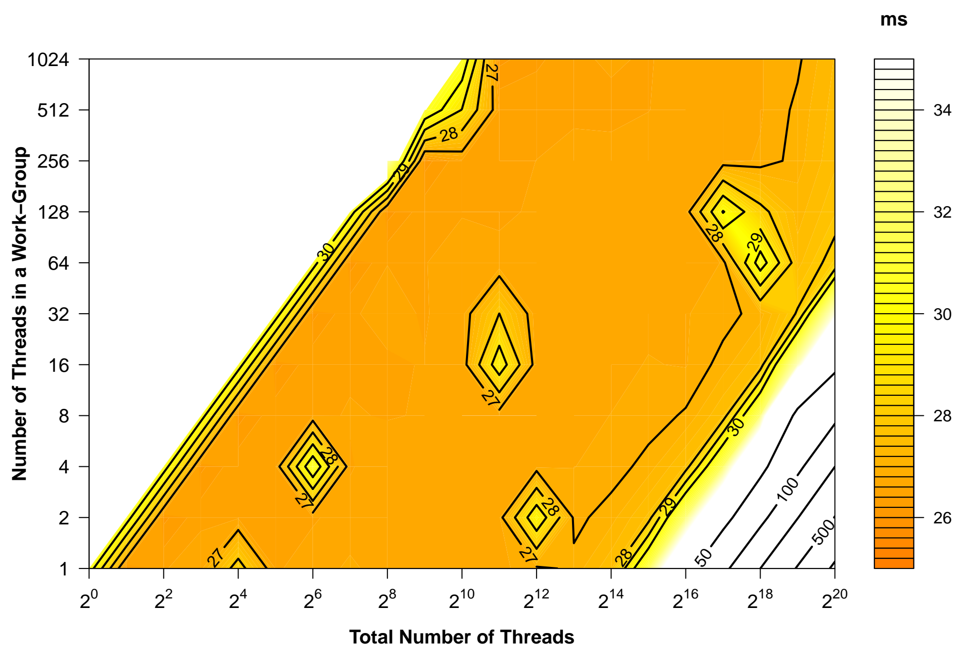


Figure B.13: Intel i3 Non-Coalesced

B.3 Compute-Adaptable Kernel

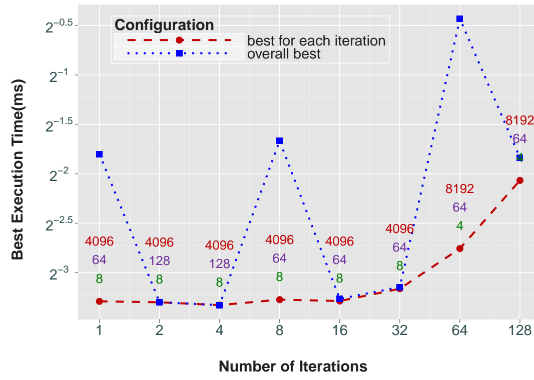


Figure B.14: ATI Non-Coalesced

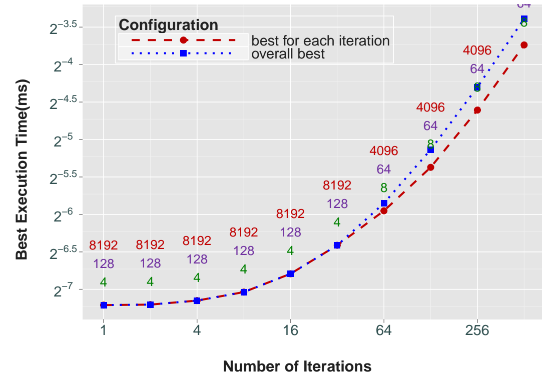


Figure B.15: Nvidia Non-Coalesced

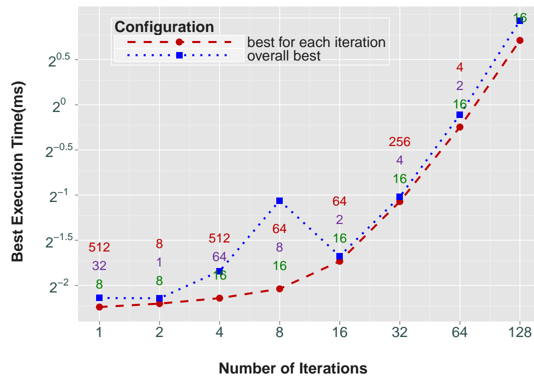


Figure B.16: Intel i3 Non-Coalesced

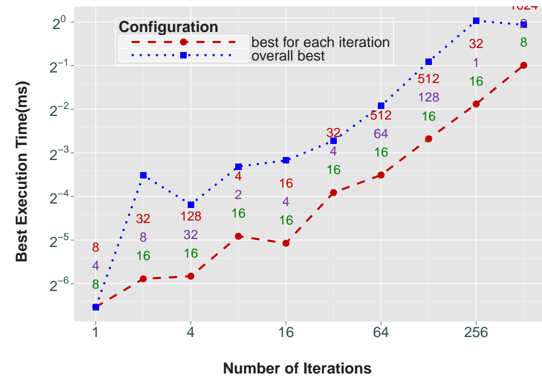


Figure B.17: Intel i7 Non-Coalesced

B.4 Unrolling Loops

In this section, the preliminary analysis of the effect of unrolling loops on the multi-core architectures is presented. For all the architectures, unrolling loops shows improvement in performance. However, if the unrolling factor is high enough, the optimization could be detrimental to performance. This behaviour is seen on both the GPUs. When the unrolling factor is 128, the execution time on the ATI becomes higher than the execution time with no unrolling.

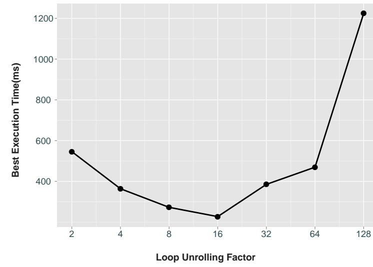


Figure B.18: ATI Coalesced

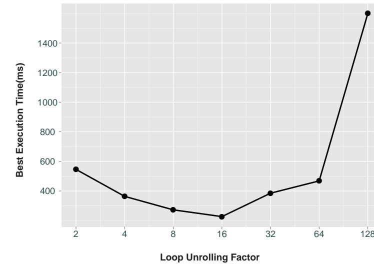


Figure B.19: ATI Non-Coalesced

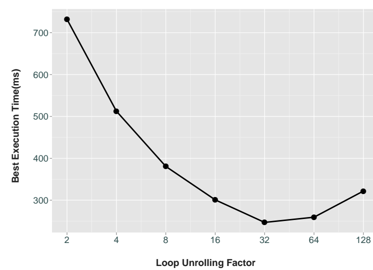


Figure B.20: Nvidia Coalesced

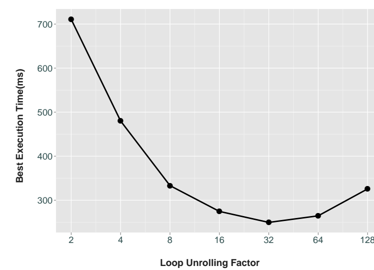


Figure B.21: Nvidia Non-Coalesced

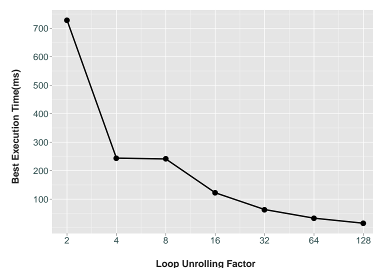


Figure B.22: Intel i3 Coalesced

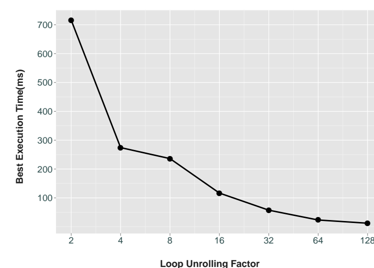


Figure B.23: Intel i3 Non-Coalesced

Similarly, for the Nvidia also, an unrolling factor of 128 leads to a decreased in performance. On the CPUs, there is no decrease in performance and the execution time keeps on decreasing as the unrolling factor increases. The experiments have been

performed keeping the other optimization parameters fixed. With more time, further analysis could be done to investigate the impact of loop-unrolling on all the other parameters.

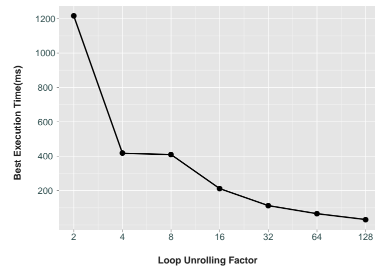


Figure B.24: Intel i7 Coalesced

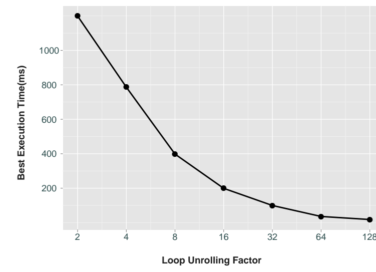


Figure B.25: Intel i7 Non-Coalesced

Bibliography

[1] Advanced Micro Devices, Inc. *ATI Stream SDK OpenCL Programming Guide*. Advanced Micro Devices, Inc., 2010.

[2] Krste Asanovic, Ras Bodik, Bryan C. Catanzaro, Joseph J. Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William L. Plishker, John Shalf, Samuel W. Williams, and Katherine A. Yelick. The landscape of parallel computing research: a view from Berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, December 2006.

[3] Nathan A. Carr, Jesse D. Hall, and John C. Hart. GPU algorithms for radiosity and subsurface scattering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '03, pages 51–59, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[4] Snaider Carrillo, Jakob Siegel, and Xiaoming Li. A control-structure splitting optimization for GPGPU. In *Proceedings of the 6th ACM conference on Computing frontiers*, CF '09, pages 147–150, New York, NY, USA, 2009. ACM.

[5] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization, 2009. IISWC 2009.*, pages 44–54, October 2009.

[6] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370 – 1380, 2008. General-Purpose Processing using Graphics Processing Units.

[7] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '10, pages 115–126, New York, NY, USA, 2010. ACM.

[8] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The Scalable Heterogeneous Computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 63–74, New York, NY, USA, 2010. ACM.

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

[10] R. Duncan. A survey of parallel computer architectures. *Computer*, 23(2):5 –16, feb 1990.

[11] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '04, pages 133–137, New York, NY, USA, 2004. ACM.

[12] M. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Trans. Comput.*, C-21:948+, 1972.

[13] GCC. GCC, the GNU Compiler Collection. `http://gcc.gnu.org/`.

[14] Mark Harris. Mapping computational concepts to GPUs. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[15] K. Hillesland and A. Lastra. GPU floating-point paranoia. In *ACM Workshop on General Purpose Computing on Graphics Processors*, pages C–8, 2004.

[16] Peter H. Hofstee. Cell Broadband Engine Architecture from 20,000 feet. http://www-128.ibm.com/developerworks/power/library/pa-cbea.html, August 2005.

[17] Intel Core i3 Spec. Intel Core i3-350M Processor Specification. http://ark.intel.com/products/43529.

[18] Intel Core i7 Spec. Intel Core i7-990X Processor Specification. http://ark.intel.com/products/52585.

[19] ICSA. Institute for Computing Systems Architecture. http://wcms.inf.ed.ac.uk/icsa/.

[20] Intel. Intel HT Technology. http://www.intel.com/technology/platform-technology/hyper-threading/index.htm.

[21] Changhao Jiang and Marc Snir. Automatic Tuning Matrix Multiplication Performance on Graphics Hardware. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 185–196, Washington, DC, USA, 2005. IEEE Computer Society.

[22] Y. Jiao, H. Lin, P. Balaji, and W. Feng. Power and Performance Characterization of Computational Kernels on the GPU. In *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, GREENCOM-CPSCOM '10, pages 221–228, Washington, DC, USA, 2010. IEEE Computer Society.

[23] KGPU. KGPU: enabling GPU computing in Linux kernel. http://gpgpu.org/2011/05/04/kgpu-gpu-computing-in-linux-kernel.

[24] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.

[25] Kazuhiko Komatsu, Katsuto Sato, Yusuke Arai, Kentaro Koyama, Hiroyuki Takizawa, and Hiroaki Kobayashi. Evaluating Performance and Portability of OpenCL Programs. In *The Fifth International Workshop on Automatic Performance Tuning*, June 2010.

[26] N. S. L. Phani Kumar, Sanjiv Satoor, and Ian Buck. Fast Parallel Expectation Maximization for Gaussian Mixture Models on GPUs Using CUDA. *High Performance Computing and Communications, 10th IEEE International Conference on*, 0:103–109, 2009.

[27] E. Scott Larsen and David McAllister. Fast matrix multiplies using graphics hardware. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '01, pages 55–55, New York, NY, USA, 2001. ACM.

[28] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 101–110, New York, NY, USA, 2009. ACM.

[29] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 451–460, New York, NY, USA, 2010. ACM.

[30] Fred Lionetti, Andrew McCulloch, and Scott Baden. Source-to-Source Optimization of CUDA C for GPU Accelerated Cardiac Cell Modeling. In Pasqua DAmbra, Mario Guarracino, and Domenico Talia, editors, *Euro-Par 2010 - Parallel Processing*, volume 6271 of *Lecture Notes in Computer Science*, pages 38–49. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-15277-15.

[31] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, April 1965.

[32] J. Nickolls and W.J. Dally. The GPU Computing Era. *Micro, IEEE*, 30(2):56 –69, march-april 2010.

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

[34] Nvidia. Nvidia GPU Computing. `http://www.nvidia.co.in/page/gpu_computing.html`.

[35] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. NVIDIA Corporation, 2007.

[36] NVIDIA Corporation. *OpenCL Programming Guide for the CUDA Architecture*. NVIDIA Corporation, 2007.

[37] Satoshi Ohshima, Kenji Kise, Takahiro Katagiri, and Toshitsugu Yuba. Parallel processing of matrix multiplication in a CPU and GPU heterogeneous environment. In *Proceedings of the 7th international conference on High performance computing for computational science*, VECPAR'06, pages 305–318, Berlin, Heidelberg, 2007. Springer-Verlag.

[38] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[39] Parboil. Parboil benchmark suite. `http://impact.crhc.illinois.edu/parboil.php`.

[40] Piotr Luszczek Stanimire Tomov Gregory Peterson Jack Dongarra Peng Du, Rick Weber. From CUDA to OpenCL : Towards a Performance-portable Solution for Multi-platform. *Parallel Computing*, pages 1–12, 2010.

[41] Matt Pharr and Randima Fernando. *GPU Gems 2: programming techniques for high-performance graphics and general-purpose computation*. Addison-Wesley Professional, first edition, 2005.

[42] Stefan Popov, Johannes Gnther, Hans-Peter Seidel, and Philipp Slusallek. Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Computer Graphics Forum*, 26(3):415–424, 2007.

[43] Sean Rul, Hans Vandierendonck, Joris D'Haene, and Koen De Bosschere. An experimental study on performance portability of OpenCL kernels. In *Application Accelerators in High Performance Computing, 2010 Symposium, Papers*, page 3, Knoxville, TN, USA, 2010.

[44] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM.

[45] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, John A. Stratton, and Wen-mei W. Hwu. Program optimization space

pruning for a multithreaded GPU. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 195–204, New York, NY, USA, 2008. ACM.

[46] Michael S. Schlansker, B. Ramakrishna Rau, and Multitemplate. EPIC: An architecture for instruction-level parallel processors. Technical report, 2000.

[47] H. Sharangpani and H. Arora. Itanium processor microarchitecture. *Micro, IEEE*, 20(5):24 –43, sep/oct 2000.

[48] ATI GPU Spec. ATI Mobility Radeon HD 5470 Specification. `http://www.amd.com/uk/products/notebook/graphics/ati-mobility-hd-5400/Pages/hd-5470-specs.aspx`.

[49] NVIDIA GPU Spec. NVIDIA Tesla C2070 Specification. `http://www.nvidia.co.uk/object/product_tesla_C2050_C2070_uk.html`.

[50] J.E. Stone, D. Gohara, and Guochun Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science Engineering*, 12(3):66 –73, may-june 2010.

[51] John E. Stone, James C. Phillips, Peter L. Freddolino, David J. Hardy, Leonardo G. Trabuco, and Klaus Schulten. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry*, 28(16):2618–2640, 2007.

[52] Samuel S. Stone, Justin P. Haldar, Stephanie C. Tsao, Wen-mei W. Hwu, Zhi-Pei Liang, and Bradley P. Sutton. Accelerating advanced mri reconstructions on GPUs. In *Proceedings of the 5th conference on Computing frontiers*, CF '08, pages 261–272, New York, NY, USA, 2008. ACM.

[53] AMD APP SDK v2.5. AMD APP SDK v2.5 Release Notes. `http://developer.amd.com/sdks/amdappsdk/assets/AMD_APP_SDK_Release_Notes_Developer.pdf`.

[54] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1999.