High Level OpenCL Implementation for Multi-core CPUs

Submitted in partial fulfilment
of the requirements of the degree
Bachelor of Science (Honours)
of Rhodes University

Matthew Royle

November 2009

Abstract

OpenCL has been developed for writing programs that run in parallel. Although OpenCL has been engineered to take advantage of GPUs (graphics processing units) it could still potentially be used on other architectures such as multi-core CPUs. This project investigates the problem of implementing OpenCL for multi-core CPUs. Implementing OpenCL for multi-core CPUs is achieved by creating a C library of OpenCL functions. The resulting parallel implementation is analyzed to have faster execution times than a comparable sequential program, but slower performance than a GPU implementation of OpenCL.

Acknowledgements

I would like to thank my supervisor, Professor Shaun Bangay, for his support, encouragement, guidance and help throughout the year.

Many thanks to my family and friends for everything they have done to help and support me, as well as the financial support afforded to me to allow me to reach this point in my career.

I acknowledge the financial and technical support of this project of Telkom SA, Comverse SA, OpenVoice, Stortech, Tellabs, Amatole, Mars Technologies, Bright Ideas Projects 39 and THRIP through the Telkom Centre of Excellence at Rhodes University.

I acknowledge the financial support of the Student Graduate Bursary extended to me by Rhodes University.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Problem Statement

The problem we solve is how we can implement OpenCL (Open Compute-Language) for multiple-core CPUs (Central Processing Unit).

Currently there is no implementation of OpenCL for multi-core CPUs. OpenCL uses a parallel execution model, which has been engineered to take advantage of massive parallel processing platforms such as GPUs (Graphics Processing Units) and their memory architecture. GPUs consist of hundreds of processing cores, which are very good at vector processing, while CPUs consist of a only a few processing cores (typically four). We investigate the issues relating to implementing OpenCL in parallel on multi-core CPUs.

To test the implementation, we analyze the performance of OpenCL using multi-core CPUs and compare this performance with the performance of OpenCL on GPUs.

1.2 Thesis Outline

This thesis consists of five chapters. These chapters are Related Work, Design, Implementation, Evaluation and Conclusion.

In Chapter 2 on page 12, we assess and summarize related work which have taken place in our field. Each piece of related work contributes to parts of this project. This provides insight into the area of work in which are participating.

Chapter 3 on page 21 is dedicated to the design of the system. During this section we analyze the intended design of the system, and its shortcomings and propose how the OpenCL design fits into our design.

Chapter 4 on page 37 explains how we implement our proposed design and what issues were involved and how we overcame those issues.

Chapter 5 on page 50 involves analyzing the performance of our implementation. We attempt to understand the effects that the system has upon OpenCL programs which are executed. We also compare the performance of our implementation compared to the performance of a GPU implementation of OpenCL created by NVIDIA. Furthermore we analyze the parallel performance of the implementation against the sequential performance using the same function. We also compare the performance of our memory implementation with the performance of the GPU implementation.

Lastly, Chapter 6 on page 66 provides a summary of our project, after which we draw conclusions on our project.

Chapter 2

Related Work

This chapter deals with previous work which has been undertaken in our field of computing. The work which is reviewed all relates to our project is some manner. This chapter is divided into six sections; Introduction, Parallel Computing, GPGPU Computing and Optimization, Heterogeneous Processing, Translators and a Summary.

2.1 Introduction

OpenCL (Open Compute Language) is a new specification which is intended for use as a heterogeneous parallel programming tool. Creating a high-level language for OpenCL will encompass different technologies related to computing. The technologies include parallel computing, graphics computing, heterogeneous processing and translators.

There is a wide variety of research which is related to the topics of parallel computing and general purpose computing using graphics processors. The area of research involving programming heterogeneous clusters is growing quite rapidly. This is the result of different architectures which have become available for use in everyday computing. Section 2.2 describes methods used to convert serial programs into parallel programs. The focus is on the use of the OpenMP API to create parallel constructs by enhancing normal serial code through the use of special directives. Section 2.3 relates to the highly parallel nature of graphics cards and optimization principles and targeting. This area of research is mainly focused on the Compute Unified Device Architecture (CUDA) developed by NVIDIA [11] and Brook+[2] developed by Advanced Micro Devices (AMD). Section 2.4 presents some architectures for heterogeneous processing on multithreaded multi-core systems. These are not necessarily in use today, but they do provide a foundation for understanding the model required to create a heterogeneous system. Section 2.5 deals with high level translators. This section provides two different examples of translators which translate a language from one architecture to another

language on another different architecture. Section 2.6 combines all the different areas of research and links them together to provide insight into creating a High-Level OpenCL Compiler.

2.2 Parallel Computing

This section deals with the writing of code optimized for parallel execution and the certain methods involved. The relevance of this section related to the creation of a high-level compiler for OpenCL is its parallel nature and how it will implement this feature using the compiler.

For many years, compilers have been directed at improving the performance of sequential programs. Due to the complex nature of modern architectures, processors have reached a point where the power they consume and heat they generate is no longer proportional to their processing power. Currently the trend is to manufacture processors which have more cores on them, rather than try and increase their clock speed. This has exposed parallelism to the programmer, resulting in the need to convert code, which was originally written as sequential code, to parallel code. To overcome this problem languages are being developed and new techniques are being used [1]. Some examples are:

- Explicitly using threads within an application
- APIs, such as OpenMP and Message Passing Interface (MPI), using directives to explicitly define parallel regions in code

Threads allow for parts of an application to be run simultaneously on a multi-core systems. Threads are common in most object orientated languages today, such as C/C++, Java and C#. Threads are implemented using a shared memory model, where threads share memory and the data stored in that memory.

Another strategy is embodied in the Application Programming Interface (API) called OpenMP (Open Multi-Processing) [3]APIs are not methods - just encapsulations of methods. The OpenMP API consists of:

- Compiler directives
- Run time routines
- Environmental Variables

OpenMP works by using the compiler directives to specify code sections which need to be separated into threads to run simultaneously. Code specified using an OpenMP directive is separated in a number of threads using the fork-join method. This results in a block of code being used for parallel processing. They also allow environmental variables to be specified which will affect how the code performs at run-time. Using this method, a shared memory model is implemented using OpenMP. [1, 3]

Techniques used on graphics cards by languages such as CUDA[11] and Brook+[2] involve the use of kernels. Kernels are written to perform a task. Kernels are each assigned to a work group or thread-block, each with an identity. Work groups are then executed. Each kernel is a logical thread in a thread block and each thread block can be executed on a processor [14]. Kernels differ from approaches such as OpenMP in the way in which they are executed. Each kernel is executed as a single thread. Executing multiple kernels results in parallel processing of shared data.

This section reveals possible approaches which can be used to implement a parallel language for OpenCL. The approach of using threads on CPUs is a simple method of achieving parallel computation. However explicitly using threads and managing them adds more work to be done in the implementation process. A way around this could be using an API such as OpenMP to automate the creation and management of threads. Out of the three approaches, OpenMP seems to be the most viable option.

The OpenCL execution model is based on the use of kernels, which are executed in a data parallel fashion. To implement this data parallel fashion, OpenMP directives can be used to specify parallel areas of execution, for example: kernels, and what is considered to be shared memory.

2.3 GPGPU Computing and Optimization

Graphics cards are being used increasingly often for executing highly parallel code. However, just like CPUs, they have limitations when it comes to certain types of applications. OpenCL is intended to overcome this problem by creating a heterogeneous parallel programming environment which can make the most use of both technologies [5]. In order to create a working OpenCL high-level language for the CPU, it is necessary to understand how code is intended for GPU processing functions and how it is best optimized.

Graphics cards were originally developed for electronic games and three-dimensional (3D) graphics [6]. However according to Halfhill [6] there has lately been a trend of using graphics cards for more than just graphics processing due to their highly parallel nature. It is stated by Boggan and Pressel [4] that the use of graphics cards pixel

shaders as general-purpose processors leads to massive parallel processing capabilities. Their design allows for parallel, computationally intensive applications which have regular memory access operations. The ability of graphics cards to be used for more than just graphics processing has led to them being referred to as General-Purpose Graphics Processing Units (GPGPU). [4, 6]

The Compute Unified Device Architecture (CUDA) was developed by the NVIDIA Corporation to simplify the process of writing programs which would run on their Graphics Processing Units (GPU). Before CUDA and Brook+ were developed, programming general purpose applications on GPUs was very difficult due to their complex nature. It required the programmer to have a very good understanding of the underlying hardware.

CUDA and Brook+ were developed to have an easy learning curve for those who are familiar with standard programming languages such as C. CUDA was developed as a parallel programming model due to the parallel nature of GPUs as described by Halfhill [6]. Brook+ was developed for stream programming using a SIMD execution model. [2, 11]

Since the release of NVIDIA's CUDA[11] and AMD's stream computing language Brook+[2], these languages have provided improved programming capabilities for GPUs. Achieving a high level of performance using CUDA or Brook+ is still challenging. This is where optimization comes to the fore.

As stated by Jang et al. [7], there are differences between graphics processing and general processing on GPUs. Users of a stream programming language like Brook+ require knowledge of the hardware to get good performance gains. Similarly Ryoo et al. [13] believe that the effort and expertise required to achieve maximum application performance is still quite high. However, the advent of languages such as CUDA and Brook+ have made the GPU platform available to more application developers than before, as well as making it less specialized. This has resulted in an increasing number of developers having access to this platform.

Languages which use are used to program GPUs, are equipped with limited optimization techniques [7]. Similarly Ryoo et al. [13] indicate that the resource restrictions in such systems present difficulties to optimization. Conversely, Ryoo et al. [12] believes that compilers such as CUDA and Brook+ increase the flexibility that developers have to tune application performance, but that they change the assumptions that developers make about optimization.

Jang et al. [7] believe that to achieve the peak performance of a multithreaded GPGPU system, a large number of active threads need to be generated to hide memory latency. Similarly Ryoo et al. [13] believe that peak performance can be achieved in three ways:

sequences of independent instructions in a warp (a group of threads in a block of threads which are executed in a Single Instruction Multiple Data (SIMD) fashion so that it can make forward progress when executing without having to wait for other threads to complete, placing many threads in a block so that at least one warp can execute and lastly to assign a thread block to each core. They describe three principles to consider when optimizing an application for the CUDA platform. These are:

- floating point throughput of an application will depend on the percentage of floating point operations contained in its instructions,
- managing global memory latency is a primary concern when trying to achieve maximum performance and
- that global memory bandwidth can limit the system's throughput.

The study by Jang et al. [7] is based on three optimization spaces, namely: Arithmetic Logic Unit (ALU) utilization, texture unit utilization and thread unit utilization. To achieve ALU utilization, they recommend using intrinsic functions provided by the Brook+ programming model whenever possible and merging sub-function calls whenever possible. To utilize texture units more effectively, Jang et al. [7] propose using the built-in short vector types in Brook+ which allows code to be explicitly tuned for an available SIMD machine. Lastly they propose better utilization of threads. It is important to have sufficient arithmetic operations versus fetch operations in each thread.

The study by Ryoo et al. [13] groups optimizations into five categories. The goal of the first category is to provide enough warps to hide stalling effects of memory latency and blocking operations. To achieve this Ryoo et al. [13] recommend decreasing thread block size and increasing the number of threads. Secondly, the redistribution of work across threads and thread blocks can lead to optimized code. Third is to reduce the number of dynamic instructions per thread. Fourth Ryoo et al. [13] believes that intra-thread parallelism ensures the availability of independent instructions within a thread. Lastly is to shift the use of resources, referred to as resource-balancing. Shifting the use of resources could involve reducing register usage, resulting in more thread blocks being assigned to multiprocessor. This could result in an application having better performance.

The study by Ryoo et al. [12] proposes using shared memory to reduce the pressure on memory latency by buffering. This can improve the access pattern for global memory. Also proposed is the use of on-chip cache, this can be implemented by using texture memory to store read-only data which can be accessed by multiple threads.

The study by Jang et al. [7] shows that applications benefit from using the different optimization spaces. They also show specific optimizations which need to be applied to the optimization spaces to obtain better performance. Their optimizations include using intrinsic functions to maximize the use of the Arithmetic Logic Unit, maximizing the use of texture units by employing long vector types and maximizing thread utilization by ensuring that there are enough active threads to hide memory access latencies.

Ryoo et al. [13] developed metrics to judge the performance of the optimum configuration. They plotted the optimal configuration on a Pareto-optimal curve, which is used to plot the best configuration for each application found after exhaustive searching. They found that this approach was able to reduce the search space for the optimum configuration by up to 98%.

Ryoo et al. [12] present an application suite, which includes their optimizations. They show that applications which have a low global memory access after optimization experience a substantial speedup over CPU execution. These results depend on applications not being limited by resource availability.

This section reveals that peak performance of an application kernel can best be achieved by optimizing the application. One general consensus is to ensure that it is important to reduce the effects of memory latency by ensuring that there are always threads executing. When creating an OpenCL high-level compiler for the CPU, it is important to take this into consideration. A CPU can execute far fewer threads than a GPU and thus these points are even more important if it is going to be possible to achieve decent performance when using a CPU.

2.4 Heterogeneous Processing

OpenCL is intended for use on multiple architectures. Creating an OpenCL high-level compiler involves the challenge creating an abstraction of the hardware so that a consistent interface is provided to the user. In this section, literature which relates to this challenge is reviewed to provide insight into different heterogeneous programming environments.

Heterogeneous processing requires processing cores of different architectures to perform operations. With the possibility of chip manufacturers such as Intel and AMD planning to integrate CPU and GPU cores into a single processor package, heterogeneous processing could become more prevalent in the near future [10, 15]. These cores can be of varying size, complexity and performance [8].

According to Kumar et al. [8], the design of modern multi-core processors must balance

the competing objectives of a high throughput versus a good single-thread performance. Kumar et al. [8] believes that processors designed to have a heterogeneous set of processor cores which execute the same Instruction Set Architecture (ISA) can deliver a greater throughput and area efficiency versus a non-heterogeneous processor. Kumar et al. [8] argues that the ability to match different applications according to their performance demands to the different core types result in a better performance. Kumar et al. [8] concludes that according to their results, a heterogeneous processor using two cores types achieves a performance gain of up to 63% over an equivalent homogeneous processor. [8]

However, current heterogeneous systems have very different Instruction Set Architectures and functionality [15]. This results in a challenge when trying to program a multi-core platform like this. Proposed by Wang et al. [15] is an architecture, called the Exoskeleton Sequencer (EXO), which aims to represent heterogeneous processors as ISA-based Multiple Input Multiple Data (MIMD) resources and a shared virtual memory heterogeneous multithreaded program execution model. Added to the architecture is a C/C++ programming environment called C for Heterogeneous Integration (CHI). The environment further extends OpenMP pragmas to allow for thread-level parallelism. Wang et al. [15] concludes by stating that their environment has improved productivity over device-driver based development environments. [15]

A framework called Merge proposed by Linderman et al. [10] is one which has been created to take advantage of the EXOCHI architecture and programming environment[15]. The framework includes a high-level parallel programming model, compiler and runtime for heterogeneous multi-core platforms. The framework works by mapping applications to a set of primitives created with something like EXOCHI. EXOCHI is an architecture and programming environment which has been created for a heterogeneous, multi-core, multi-threaded environment [15]. Computations are expressed using high-level language extensions which are architecture-independent. Linderman et al. [10] concludes that the framework has the potential to replace ad hoc approaches to parallel programming on heterogeneous systems with a library-based methodology. [10]

This section provides some ideas which can be applied to creating an OpenCL high-level compiler. A common theme is the extension of a current environment using a high-level language. Creating libraries which can be used on a system of varying architectures will help to simplify the process of creating a compiler.

2.5 Translators

The OpenCL high-level compiler is essentially acting as a translator of code. This process involves converting OpenCL code to C code and then compiling it using a C

compiler. This section reviews translators which take code for a certain architecture, for example: GPU, and then compile it to run on another architecture, for example: CPU. This section is relevant in that OpenCL is intended for use on multiple environments, but this project initially encompasses creating a compiler for the CPU architecture only.

Translators are used to convert a code from one language to make it compile and run on other languages. These languages can be ones created for a specific parallel architecture, for example CUDA[11]or Brook+[2] for GPUs.

Lee et al. [9] proposes a compiler framework which will translate code using OpenMP to code which will run on GPGPUs using NVIDIA's CUDA language. According to Lee et al. [9] it is more complex and in some cases more difficult to create code for a GPGPU using CUDA. Thus the reason for creating a compiler framework which converts code produced using OpenMP to code which will run using CUDA for a GPGPU is that producing code using OpenMP takes less effort. The framework works in phases; phase one involves an OpenMP stream optimizer which makes code created for the CPU more GPU friendly. The second phase involves converting OpenMP code to GPGPU code using a translator. Phase two involves two separate steps, the first step is identifying kernel regions based on OpenMP parallel regions. The next step in phase two is transforming the identified kernel regions into a kernel function. During step two in phase two, data mappings are performed. All of the data is mapped to global memory regardless of their visibility. Even though OpenMP code is suitable to be converted to run on GPUs, it was found that it does not always result in good performance. This was the case for both regular and irregular applications. However, Lee et al. [9] found that the performance achieved by the framework came close to that of hand-coded CUDA programming.

Another proposed translator, which works in the opposite direction is called MCUDA [14]. This is an attempt to efficiently implement CUDA kernels on multi-core CPUs. To achieve this, a thread block needs to be transformed into a serial function. The threads used on GPUs are different to those used on CPUs. Another issue to deal with is memory spaces, memory spaces are used differently on GPUs than compared to CPUs. GPUs use differentiated memory space, while CPUs use unified memory space. This is another issue that needs to be dealt with to get CUDA kernels to run on the CPU. Stratton et al. [14] achieves this by changing the nature of the kernel from per-thread code specification to a per-block code specification. The next step involves enforcing synchronization in the kernel code to ensure that the kernel executes in order. Lastly the system is required to replicate thread-local data. Stratton et al. [14] achieves this by creating private storage for each instance of the thread's variable. Arrays are also created for local variables when necessary to use less memory. Since CPUs and

GPUs are vastly different, it is to be expected that this system would not achieve the same performance as the kernel run on a GPU. Stratton et al. [14] found that the performance of MCUDA was significantly lower than that of good hand-tuned CUDA code. Stratton et al. [14] tested their implementation by performing three tests; matrix multiplication, Coulombic Potential and high-resolution MRI image reconstruction.

This section provides insight into the process of creating a translator. There are techniques which are used here that would be relevant to the process of creating an OpenCL high-level compiler. This section verifies that it is possible to translate code intended for one architecture to another architecture.

2.6 Related Work Summary

There is a common theme which relates to the work reviewed in this chapter, this is parallel processing. Parallel processing is a growing trend amongst programming languages and architectures. For the purpose of this project, we gain insight into emerging trends in computing and the potential processes and problems involved in creating a high level OpenCL implementation for multi-core CPUs.

Some conclusions which are made in this chapter involve threading in programming languages, heterogeneous processors and code translators.

The trend of increasing the number of processing cores on processors has led to the need for methods which can take advantage of this trend. A common method used, is making use of threading in programming languages.

Architectures which make use of heterogeneous processors have been found to have improved performance over non-heterogeneous processors. This discovery has been attributed to their ability to scale different programs to different processors. However it is also important that the language used is implemented to take advantage of the architecture.

The process of translating code designed for one architecture to another can be achieved. To achieve the translation process, the differences in architectures needs to be considered. This is important since languages are designed to take advantage of certain architectures, those advantages sometimes do not apply to other architectures.

Chapter 3

Design

This chapter explores the design of OpenCL [5] in relation to parallel architectures and how it maps to these parallel architectures. This chapter intends to differentiate the parallel models and how this affects the mapping of OpenCL to different architectures. W discuss these differences and how these affect implementing OpenCL for multi-core CPUs.

In Section 3.1 we discuss some background behind OpenCL, this includes an abstract parallel model which introduces components which make up a parallel processing architecture.

In Section 3.2 we discuss how the OpenCL design maps onto the GPU parallel architecture and how this differs from mapping OpenCL to a CPU parallel architecture. We consider how these differences could potentially affect the implementation of the OpenCL design. The aspects of the design which we consider, includes the design of devices and contexts, buffers, kernels and the execution model.

Section 3.3 summarizes the OpenCL design and discusses the overall effect of designing OpenCL to be used with a parallel CPU architecture rather than the intended GPU parallel architecture.

3.1 OpenCL Background

Because OpenCL is a parallel processing language, we need to consider what constitutes a parallel architecture. This is important as we need to understand how OpenCL has been designed to take advantage of the parallel architecture. We also need to understand how the components, which make up the OpenCL language, map to the architecture. For the design of the system, we assume an abstract parallel architecture as depicted in Figure 3.1. In this architecture, we have a set of processor nodes, each

of which are each capable of performing some sort of calculation. Calculations can be performed concurrently with other processing nodes, while being independent of the other processing nodes. Each processor node has access to its own private memory along with access to global memory.

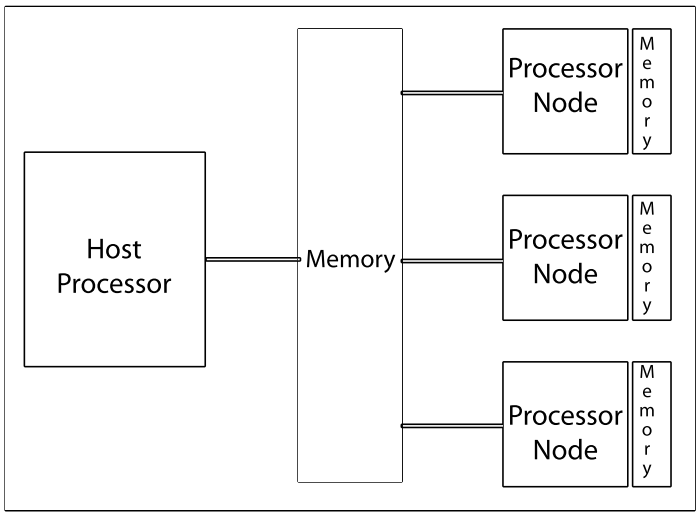

Figure 3.1: Abstract Parallel Architecture

Data are stored in buffers which are created in memory. This data can be used in calculations or can be the result of calculations performed by the processor nodes. This system adopts a Single Program Multiple Data (SPMD) processing approach. This works by each processor node executing the same program, with each execution instance of the program using a different region of the data set as its own set of data input.

Also present is a host processor which acts as a control device for the system and can be used to analyze the results once execution has taken place. As a result of this, the host processor also has access to global memory. We assume that the host processor also has some form of private memory, but this is largely irrelevant in this parallel architecture as it does not affect the functioning of the system directly.

3.2 OpenCL Design

OpenCL has been designed as a heterogeneous parallel processing language. As such we assume that the hardware architecture of a system, on which OpenCL code is executed, can include a number of different types of processors performing calculations concurrently. However, OpenCL has been specifically designed to take advantage of the massive parallel processing power of graphics cards which are used as GPGPUs. For

this reason we must consider a parallel architecture which includes GPUs as parallel processors in the system. This architecture makes use of the components introduced by the parallel architecture illustrated in Figure 3.1. The GPU parallel architecture on which the OpenCL design is based on is depicted in Figure 3.6. To understand the design, it is important to consider the GPU architecture and how the OpenCL design maps to this architecture.

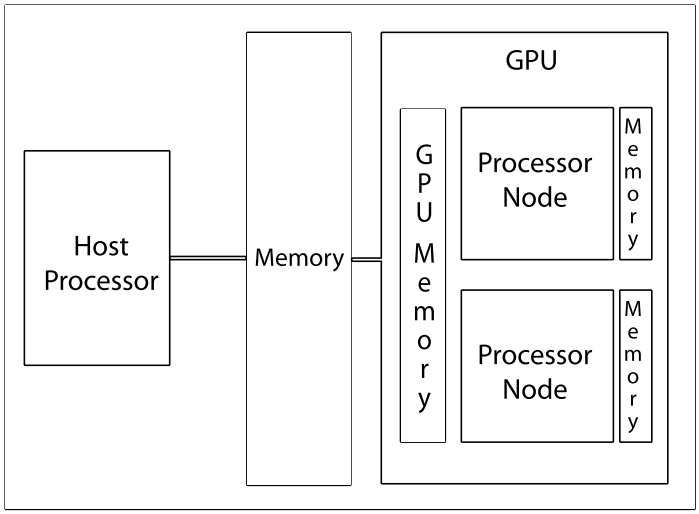

Figure 3.2: GPU Parallel Architecture

In a GPU parallel architecture, the host processor is defined as a CPU in the system. The memory in the system consists of three different memory types, the first is the Random Access Memory (RAM), or primary memory, of the system. This memory can be accessed and used by any device in the architecture, for this reason it is referred to as global memory. Each device also has its own set of memory, or GPU memory. This memory is made available to any of the processing nodes in the device, for this reason it is referred to as shared memory. Finally, the last memory type is only available to a processor node and often resides on the node itself. For this reason it is referred to as local memory. The processor nodes are defined as a set of processors found on graphics cards (GPU) in the system. These processor nodes have the ability to perform parallel computations very rapidly. However the manner in which they are programmed to perform these computations differs greatly from the manner in which CPUs are programmed to perform computations. For this reason, the design of OpenCL has functionality which is specifically aimed at utilizing GPUs as the parallel processing nodes in the system.

GPUs require specialized languages provided by their manufacturers to program them. These languages are usually unique to the card manufacturer and potentially the GPU model itself. An example of such a language is CUDA, which has been developed for

use on graphics cards which are developed by NVIDIA[11].

CPUs differ from graphics cards because they can be programmed using many different languages. These languages may take different approaches to programming. An example is functional languages which are designed differently from object-orientated languages.

Since the aim of our project is to implement OpenCL for multi-core CPUs, we need to consider a CPU parallel architecture and how we map OpenCL to this architecture. The CPU parallel architecture is illustrated in Figure 3.3. We need to explore the differences between mapping OpenCL to the GPU parallel architecture and mapping OpenCL to the CPU parallel architecture and how we modify the OpenCL design for the CPU architecture.

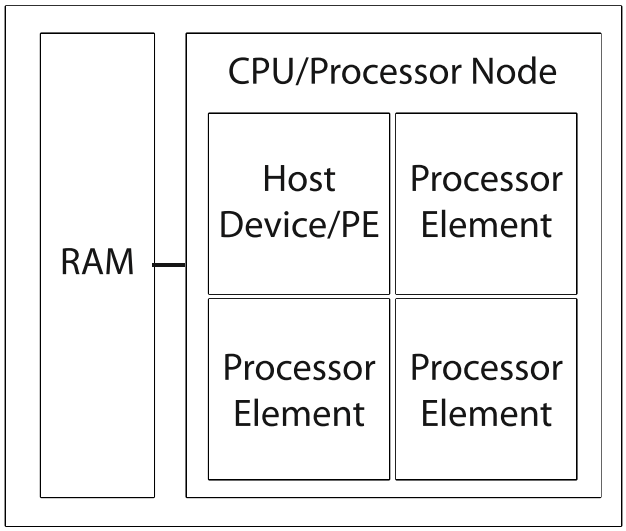

Figure 3.3: CPU Parallel Architecture

In a CPU parallel architecture, we only consider there to be one device in the system, which is the CPU. Within the CPU, one processing core acts as the host device. Although a single core acts as the host device, it can also be used as a processing element (PE) to execute programs. The CPU itself is a processor node as well as being a device in the system. The other CPU cores are used to execute programs. They can be used to execute programs simultaneously resulting in parallel execution. In our architecture, we have only one instance of memory available to the system. This is the primary memory, or Random Access Memory (RAM), of the system.

There are four main aspects of the OpenCL design to consider when mapping the design to different architectures. These are:

- Devices and Context

- Buffers
- Kernel
- Execution Model

These are central to the manner in which OpenCL is supposed to function. For this reason we need to find out how they are intended to map to the GPU parallel architecture and how we can map them to the CPU parallel architecture.

3.2.1 Devices and Context

Creating an OpenCL system which makes use of multiple processors which are potentially heterogeneous leads to the need for a system which can distinguish one processor from another. For this reason, devices in a system need to be uniquely identifiable. Uniquely identifying the individual devices of the system allows OpenCL devices to be used independently of one another. In OpenCL, a list of devices is used to contain processing devices which are part of a system.

Furthermore we need to be able to identify which components of a computer system are also considered to be part of an OpenCL system. To allow us to achieve this, an OpenCL context needs to be defined. A context can be used to identify the scope of the system, in relation to the rest of the system. In other words, everything which is used in an OpenCL system is considered to be part of the context.

3.2.1.1 Context

The context of a parallel processing system includes all the devices of the system along with the memory used by the system. In terms of our GPU parallel model, the context includes the host processor, the main memory and the GPU along with all of its individual components. The individual components of the GPU include the GPU memory, and the processor nodes along with their memory. An OpenCL context of a GPU parallel architecture is illustrated in green in Figure 3.4.

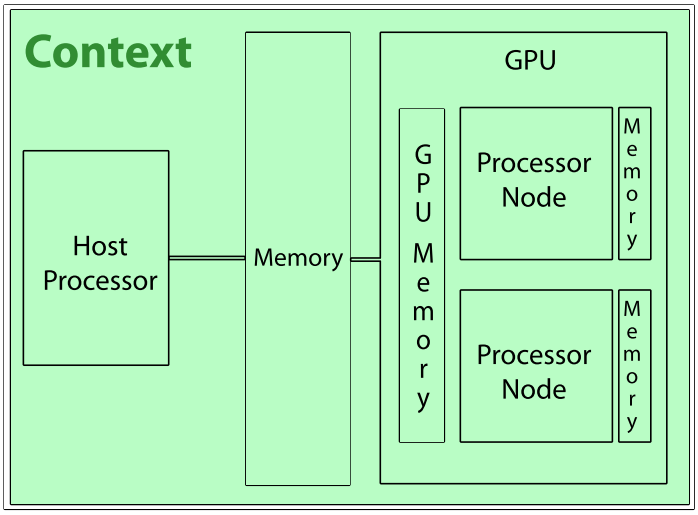

Figure 3.4: A GPU Parallel Model Context

When mapping the OpenCL context to our CPU parallel architecture, there are fewer components of the parallel architecture which are included in the context. The context of this architecture includes the primary memory and the CPU along with its individual processing cores. An OpenCL context of a CPU parallel architecture is illustrated in green in Figure 3.5. There is clearly a difference between a context which maps to a GPU parallel architecture and a CPU parallel architecture. The difference being that a context of the GPU parallel architecture has separate host devices and computational devices. A context of the CPU parallel architecture contains only one device; the CPU.

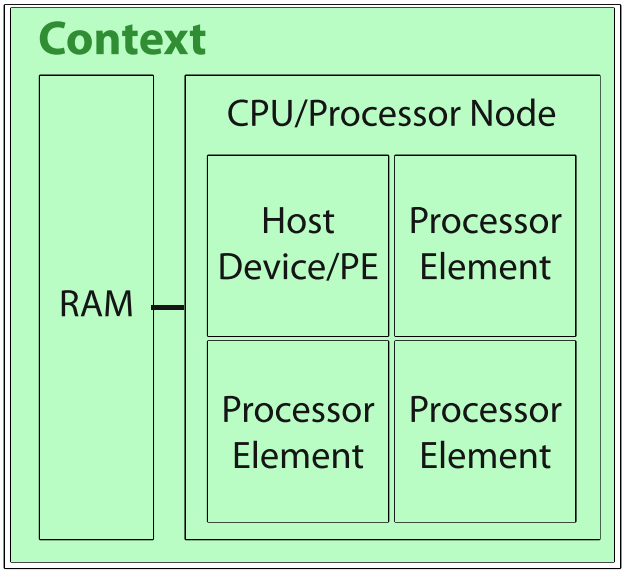

Figure 3.5: A CPU Parallel Model Context

To define the context for this system, we need information of all the available resources. This information would include all the devices and the memory available to the system,

which can be used by the system. There is the possibility that certain devices on the system are present, but cannot be used for the purpose of parallel processing.

3.2.1.2 Devices

A device is considered to be any processing device which is capable of performing general computation in a parallel environment. Devices in our architecture, illustrated in Figure 3.1 on page 22, include the host device and computational devices which are also known as processor nodes. Processor nodes can have more than one element which is capable of executing a program on its own. These elements are known as processing elements.

A host device is responsible for maintaining overall control of an OpenCL program and is charged with instantiating any components required. It is also responsible for enqueuing commands to be executed. The processor nodes are responsible for executing kernels. They are used to create the parallel nature of the system.

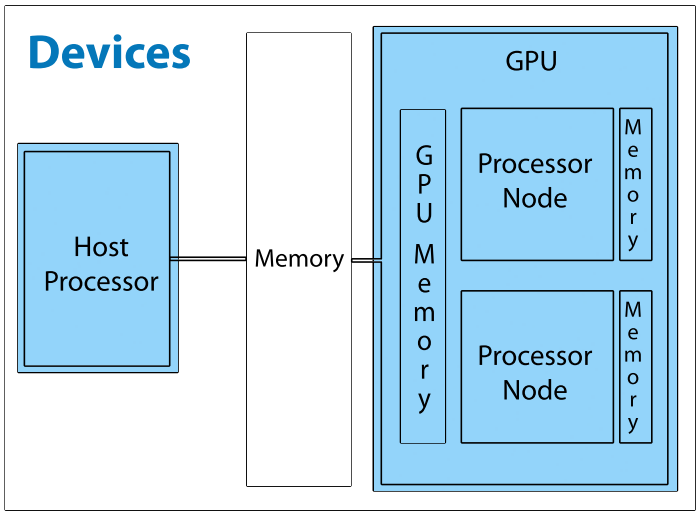

Figure 3.6: GPU Parallel Model Devices

When considering the GPU parallel architecture, there are two distinct devices which can be identified. These are the host device and the GPU. These devices are highlighted in blue in Figure 3.6. In the GPU parallel architecture, the CPU is always considered to be the host device, while the GPU is used as the computational device. The parallel nature of the architecture is contained within the GPU and how it performs computations.

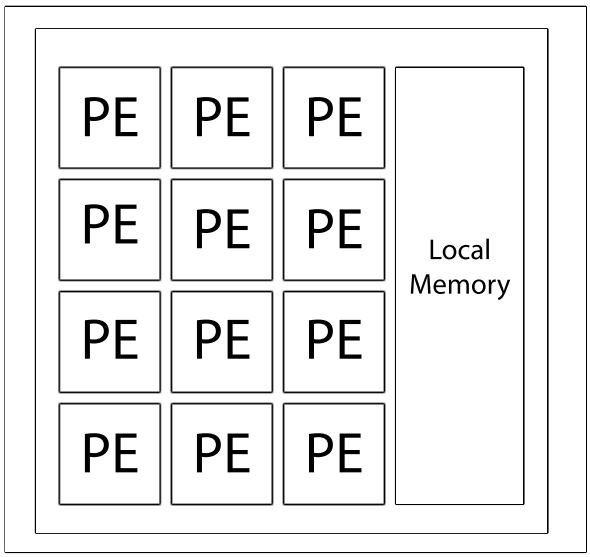

Figure 3.7: GPU Processor Node

A GPU processor node contains a number of processing elements. Processing elements are illustrated in Figure 3.7. Elements are able to process tasks simultaneously with the tasks being independent from one another. Because GPUs possess a large number of processing elements, they are able to process a large number of tasks simultaneously resulting in massive parallel processing.

Devices in the CPU parallel architecture differ from the GPU parallel architecture in that there are fewer devices. These devices are highlighted in blue in Figure 3.8. There is essentially only a single device in the architecture. The CPU acts as both the host device, while also being responsible for processing tasks. This is different from the GPU parallel architecture as there is only one device in the CPU parallel architecture.

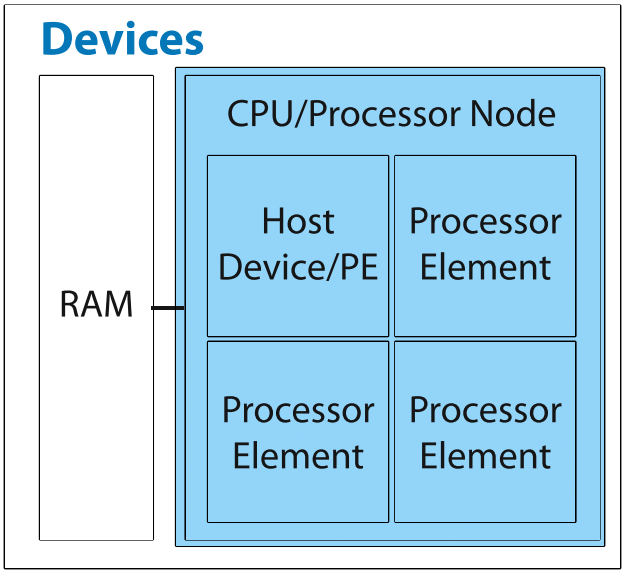

Figure 3.8: CPU Parallel Model Devices

OpenCL devices are intended to have a command queue associated with them. Command queues allow commands to be en-queued which need to be executed by the associated device. Commands can include executing programs, reading data from a buffer or writing data to a buffer.

3.2.2 Buffers

Buffers are used to read data from memory and write data to memory. There are three different types of memory available, these are:

- Global memory
- Shared memory
- Local memory.

Memory in the GPU Parallel Model is highlighted in orange in Figure 3.9. Illustrated in this figure are three distinct regions of memory. Each region of the memory in the GPU parallel architecture is associated with a different memory type.

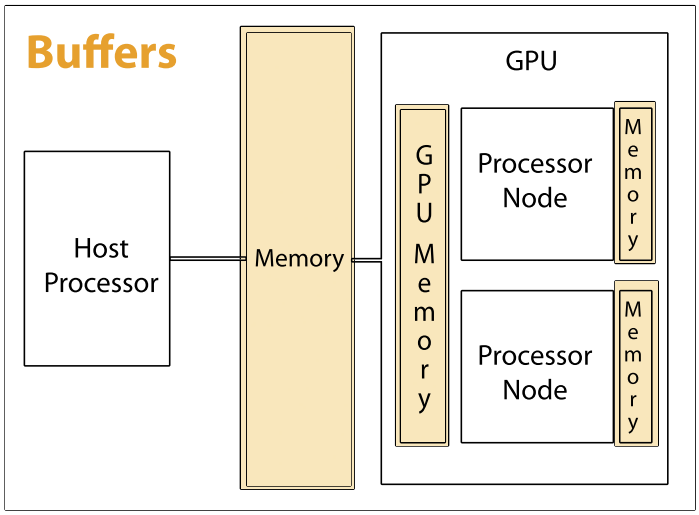

Figure 3.9: Memory in a GPU Parallel Model

Highlighted in Figure 3.10, is the memory in a CPU parallel architecture. When looking at the distinct regions of memory in a CPU parallel model, we observe that there is only a single region of memory available. As a result of this the types of memory will all be associated with the single region of memory.

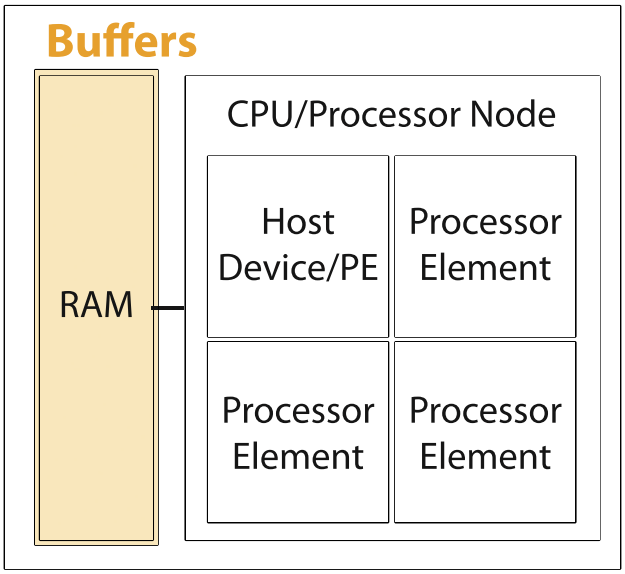

Figure 3.10: Memory in a CPU Parallel Model

3.2.2.1 Global Memory

Global memory is intended to be accessible by any device which is included in the context. In the architecture assumed in Figure 3.9, global memory is the primary memory (RAM) of the system. In the architecture assumed in Figure 3.10, global memory is also the primary memory of the system. As a result of this, global memory is identically mapped to the two architectures.

As this memory is accessible by all the devices included in the context, this memory is typically used to store a complete data set which is being worked with. As parts of the data set are needed by a device, then that part is copied to another buffer. OpenCL defines commands which are used to read from and write to buffers. These commands are used in conjunction with a command queue to perform buffer reads or buffer writes.

3.2.2.2 Shared Memory

Shared memory is memory which is shared by the components of a device. However it is only intended to be accessible by the components of that device. In Figure 3.9, shared memory is the GPU memory included on the GPU.

In Figure 3.10, there is no distinct memory region associated with shared memory. As a result of this, shared memory is also associated with primary memory. There is a distinct difference in the mapping of shared memory to the two architectures.

3.2.2.3 Local Memory

Local memory is only accessible to a specific processor node and it's individual processing elements. This can be used as working memory while a specific node is being used. Because local memory is associated with a processor node, kernels which are executed by elements in a node have access to local memory.

In Figure 3.9, local memory is the memory associated with each processor node.

In Figure 3.10, there is no distinct memory region associated with local memory. Because of this local memory is associated with primary memory. Upon analyzing the mapping of local memory to the different architectures, it is apparent that there is a distinct difference in the mapping of local memory.

3.2.3 Kernel

A kernel is used to represent a function which needs to be computed. Kernels are intended to take advantage of the massively parallel nature of GPUs. Multiple kernels can be used for computation within a processor node since there are multiple processing elements each capable of executing a kernel. The result of this is that many kernels can be executed simultaneously using the GPU architecture.

To maximize the use of the GPU architecture, kernels are separated into work groups. Each work group contains from one to four kernels, with each kernel considered a work item. Since a work group can only contain a maximum of four kernels, kernels are uniquely identified by a combination of their work group (local) identifier and their global identifier. At execution time, an entire work group can be executed by a processor node. Kernels are executed by a processing element on a processor node. In Figure 3.11, a single processing element is illustrated in purple.

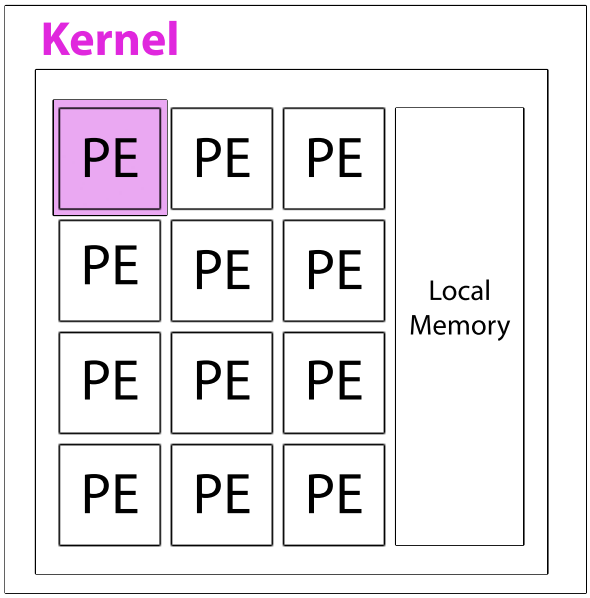

Figure 3.11: GPU Processing Element

Using work groups with the CPU parallel architecture does however have some shortfalls when trying to execute kernels. The reason for this is that the maximum number of kernels which can be executed at a time is limited to the number of processing elements which are available in the system. If we consider the illustration in Figure 3.12, we can observe that the CPU parallel architecture has fewer processing elements than the GPU parallel architecture, illustrated in Figure 3.11.

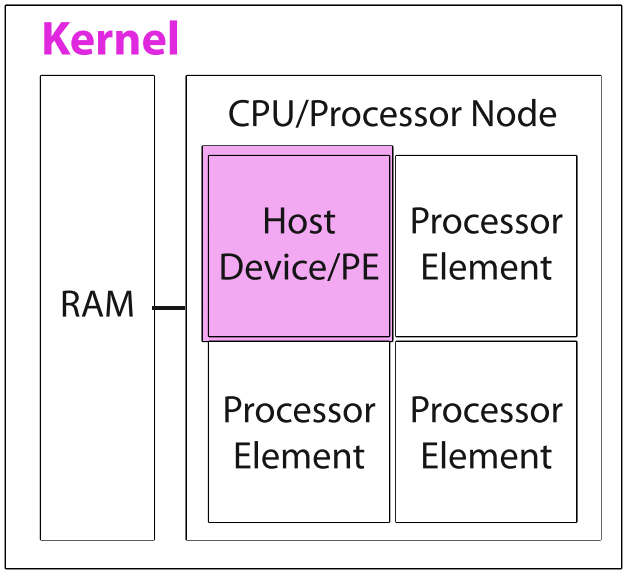

Figure 3.12: CPU Processing Element

Kernels are able to use all three memory types. However, individual kernels do not work on the entire data set stored in a buffer. Kernels are given a subset of the data

which they use to accomplish their task. As such they only have access to a portion of the buffer which stores the data. They use their global and local identifiers to access the portion of memory with which they are required to accomplish their task.

Before kernels can be executed, a program is required to be built from the kernel and the kernels arguments need to be set. These processes of building a kernel is described in Section 3.2.3.1, while the setting of arguments is described in Section 3.2.3.2.

3.2.3.1 Building a Kernel

.To build a kernel program, we use a temporary file which is used to hold the kernel source and a function which is used to pass arguments to the kernel when executing it. We refer to this function as a calling function. The process of building a kernel requires a couple of steps. These steps include:

- Getting the kernels header details
- Getting the parameter details of the kernel
- Using the kernel header details and parameter list details to create a temporary file
- Compiling the temporary file

3.2.3.2 Kernel Arguments

The last step before kernels can be executed requires that kernel arguments are set in the host program.

Kernel arguments can have a visibility identifier to specify what their level of visibility is when used by a kernel. These identifiers are related to the three memory types mentioned in Section 3.2.2. It is also possible for kernel arguments not to have a visibility specified. The difference between specifying a visibility for an argument or not specifying a visibility relates to the manner in which they are intended to be used. An argument with a visibility is a buffer which is being passed to the kernel, buffers are intended to have data read from or data written to them by kernels. Arguments without a visibility specified are passed as values which are used by the kernel, but are never altered in any way.

3.2.4 Execution Model

The execution model involves executing kernels in parallel. To achieve this, a kernel is executed by a single processor element. To further extend the execution model, kernels

can be placed in work groups. A work-group can be executed by a processor node. To execute kernels, command queues are used to en-queue execution commands for a device.

Executing kernels is discussed in Section 3.2.4.1, parallel execution is discussed in Section 3.2.4.2 and command queues are discussed in Section 3.2.4.3.

3.2.4.1 Executing Kernels

As stated in Section 3.2.3 on page 31, kernels represent a function which needs to be computed. They are executed using the processing elements of a processor node. OpenCL's execution model has been designed to use the SPMD parallel execution model, which allows kernels to be executed in parallel. Since the same kernel is used to perform each iteration of the execution, kernels can be executed both in-order and out-of-order by making use of command queues. Although we have a single program, or kernel, to compute our results, kernels use different regions of the data set to obtain different results.

3.2.4.2 Parallel Execution

To achieve parallel execution, one kernel is executed by one processing element at any time. With the GPU parallel architecture, this equates to multiple kernels being executed at any given time. To further enhance parallel execution, kernels are placed into work groups. Each work group can be executed by a processor node. This involves the group being sent to the processor node for execution, Within the group each kernel is assigned to a processing element to be executed. A work group sent to a processor node for execution is illustrated in red in Figure 3.13. Using this work group, four kernels can be executed simultaneously by the processing elements.

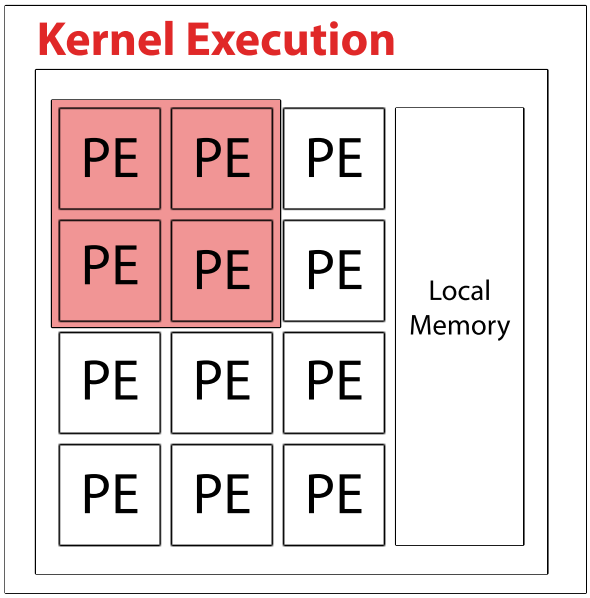

Figure 3.13: GPU Kernel Work Group

The execution model of OpenCL has been designed around this principle to take advantage of GPU parallelism. The same setup does not gain any advantages when being executed on a multi-core CPU. As illustrated in Figure 3.3 on page 24, a multi-core CPU has far fewer processing elements than a GPU. The result of this is that fewer kernels can be executed at any time. Because of this, we can do away with using groups of kernels and rather just use individual kernels for execution on a processing node. This is illustrated in red in Figure 3.14, where a work group is executed by a processing element. This equates to executing a single work group being executed on a processing element.

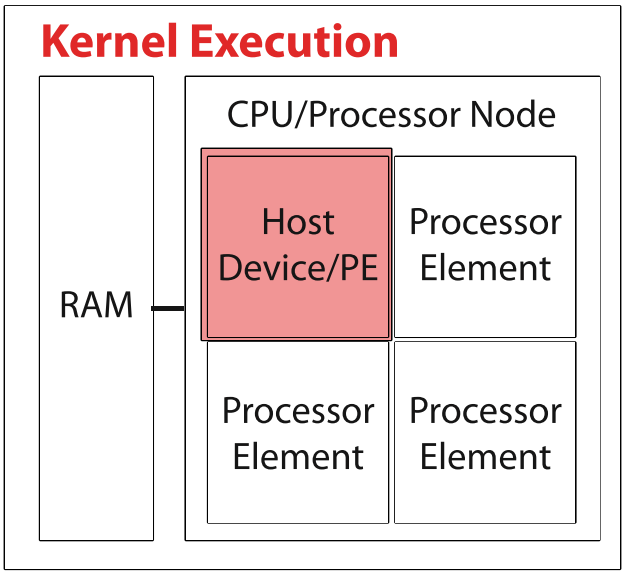

Figure 3.14: CPU Kernel Work Group

3.2.4.3 Command Queues

Kernels are executed using command queues associated with a device. When a kernel needs to be executed, an execution command is placed onto a command queue. To execute a command, OpenCL makes use of events which are linked to a command. Events allow the host program to specify a command to be executed. Events contain detail which can be used to discover the status of execution, the command type or the command queue associated with the event. It however is possible to execute commands without using events. This is achieved by first en-queuing all required commands on the command queue and then executing the commands in the order they were en-queued in. By allowing these two different approaches, OpenCL allows both in-order and out-of-order execution. Command queues can also be used to en-queue commands for reading from and writing to buffers.

3.3 Design Summary

It is clear from the OpenCL design that it has targeted GPU parallel architectures as its major method of executing programs in parallel. For this reason, the design has several aspects which have been designed in a way which is intended to take advantage of this architecture. Some of these aspects include the use of devices and contexts, a layered memory hierarchy, the use of a kernel to represent a task and the execution model. Because of this, there are aspects of the design which are difficult to map to a CPU parallel architecture. As such the design of OpenCL needs to be adapted to the CPU parallel architecture, resulting in some aspects of the design being modified to better suit the architecture.

Chapter 4

Implementation

As stated in Section 1.1 on page 10, we implement the OpenCL design for multi-core CPUs. As such we assume an architecture as illustrated in Figure 3.3 on page 24. Since we are using CPU cores as our processor nodes, we can use an existing programming language to implement the OpenCL API using a language which can be natively executed on CPUs. OpenCL is intended to be a "C" like language, for this reason we implement a set of OpenCL functions in a library using the C/C++ language. This does however raise some issues regarding parallel processing and use of all the processing nodes which are available to us. To make use of multiple nodes, we use threading to effect execution on multiple cores of the CPU. To enable the easy use of threading, we use the OpenMP API [3] to execute our kernels in parallel.

4.1 Devices and Context

The context of our system is considered to include all the parts of a system necessary to run normal C code. As such, our context includes our primary memory along with our CPU. This is illustrated in Figure 3.5 on page 26. In the architecture, the CPU is considered to be a processor node. Within the CPU, the four cores are the processing elements with one of the cores also acting as the host processor.

4.1.1 Context

Since our implementation only includes the CPU as an OpenCL device along with other system resources which are generally available to any programming tasks, we choose not to implement a proper instance of a context. Implementing a context in this manner results in our implementation only being able to make use of the CPU as a processing device and our primary memory as the memory in the system. Future extensions to the

project can include implementing a working context able to contain multiple devices and multiple instances of memory. For correctness of program structure and use of the API, we implement a context as an empty data type.

Any API calls involving a context only return a success code to ensure that the system carries on functioning correctly. However, if any of the arguments passed to the context do not meet the criteria as specified by the OpenCL Specification [5], then an error code is returned. There are two error codes we deal with, these are an invalid platform and an invalid value.

The invalid platform error code has one case we deal with. This case is when the context properties are set as null then we return an invalid platform error code.

The invalid value error code has two cases we deal with. The first case is when the device id list is null or the number of devices is set to zero then we return an invalid value error code. The second case we deal with is when the callback function is set to null, but the user data is not null then we return a an invalid value error code.

4.1.2 Devices

Since we only allow CPU cores belonging to a single system to execute our programs, the only information which we gather from the system is how many execution cores are present in the system. This is due to our decision to create a library of OpenCL calls rather than create an actual OpenCL compiler. Since we do not need to get a list of the devices on a system, we simply set the number of devices on the system to one. Using the information gathered on the system, we set the number of threads which our implementation uses for the execution of kernels.

Future extensions to the project can include getting the number of devices in the system and getting these device's details. Details could include the processor type, the clock-speed, make and model and device type.

By hard-wiring the number of devices on the system we force users to only use the CPU on a system to execute OpenCL code and does not represent an actual heterogeneous parallel processing language. Another future extension could be to allow users to make use of multiple heterogeneous devices on the system for processing.

4.2 Buffers

Since we are using a CPU parallel architecture, we model the three memory types using primary memory. The reason for this is twofold; we have no direct access to the cache

on the CPU, which could potentially act as private memory, and there is no other memory type available on the system.

We divide the buffer implementation into three distinct sections:

- Implementing a Buffer
- Reading to- and Writing from a Buffer
- Global, Local and Shared Memory

How we specified a memory type in our implementation is discussed in Section 4.2.1.

How we implemented reading to a buffer and writing from a buffer is discussed in Section 4.2.2 on the following page.

How we implement global, local and shared memory in our system is discussed in Section 4.2.3 on the next page.

4.2.1 Implementing a Buffer

To create a buffer, we require a definition for a memory type which will be used to contain the address of data in memory. This definition is illustrated in Figure 4.1. Our definition of a memory type includes a pointer. This points to an address in memory where the data in the memory type are stored.

```
struct _cl_mem
{
        void * data;
};
```

Figure 4.1: Memory Type Definition

To create a buffer in our system we use our definition of our memory type to represent an instance of a buffer. When a buffer is created, it is allocated memory from primary memory.

The buffer is used to pass data to kernels for execution or can act as shared memory between processor nodes on a device. More specifically, buffers can be used to direct input to kernels, receive output from kernels or do both.

4.2.2 Reading to- and Writing from a Buffer

Having a pointer to data stored in primary memory allows us to read data from a buffer and write data to a buffer. To achieve this we use the `memcpy` function within our OpenCL read and write buffer functions.

Reading data from a buffer allows the host to retrieve the results of a computation which has taken place. When reading from a buffer, the buffer is passed to the appropriate method as the source of data to be read. This can then be read to some output for display or further use.

Writing data to a buffer allows the host to write data which is used by kernels for computation. When writing to a buffer, a buffer is passed to the appropriate method as the destination for data to be written to. This allows data to be written to the buffer.

4.2.3 Global, Local and Shared Memory

As mentioned previously in Section 4.2.1 on the previous page, buffers reside in primary memory and are considered to be part of the global memory of an OpenCL system. For this reason buffers will be used to contain the entire data set used by a program.

Shared memory is considered to be similar to global memory and also resides in primary memory. As such we do not go to any great lengths to differentiate shared memory from global memory and use them in our implementation in the same manner.

Local memory is considered to be local to each kernel group and thus can only be "seen" by that kernel group. Often local memory contains no data when it is instantiated and is considered to be a different case to global and shared memory. This case is dealt with when setting kernel arguments, this can be found in Section 4.3.4 on page 44. Although this memory type is local to each kernel, we still use use primary memory to store its data. This is as a result of the system only possessing one set of memory which we can access and make use of.

4.3 Kernel

Kernels are used to represent a task and are the only part of an OpenCL system which actually represents any sort of computation in the system. All the other components exist to support the execution of kernels. Implementing kernels to work using our system without the use of a compiler requires some cunning processes to ensure that an OpenCL kernel can be compiled and executed on a CPU.

Before kernels can be executed, their arguments need to be set. This is another aspect which needs to be dealt with in our implementation.

Kernels are unique in that they are declared in an OpenCL file, while the rest of the API functions are generally written and executed from a C/C++ file. Because of this we require some method of compiling a kernel which can be used with our implementation. To achieve this we make use of a temporary file. To be able to create the temporary file we need to acquire certain kernel details, these details include the kernel header and its list of parameters.

The temporary file contains the kernel along with a calling function. Because kernels differ from one another, we have no way of instantiating a kernel at run-time without hard-coding them individually. For this reason we make use of a calling function which is unique to each kernel. The calling function is called when kernels need to be executed. It is used to pass arguments to kernels and call the kernel as a function so that it can be executed. When it calls the kernel, it sets the kernel's global identifier so that it can use the correct data set for its computations. The temporary file is created and compiled as a C file.

The entire process of compiling a kernel involves two API function calls. The first function is used to create a program using the kernel as a source string. Acquiring the kernel source string is left up to the programmer to do. This function is used to acquire details of the kernel. The second function is used to create the temporary file which contains the kernel source and its calling function. Once the temporary file has been created, it is compiled. The process of acquiring kernel details is divided into acquiring kernel header details and acquiring kernel parameter list details. These processes are discussed in Section 4.3.1 and in Section 4.3.2 on the following page respectively. The process of building a kernel is discussed in Section 4.3.3 on page 43.

Once a kernel has been compiled, we need to be able to store the kernel's arguments before it can be executed. Without doing this we would not be able to pass arguments to the kernel. This is because kernels are not intended to be used like normal functions in programming, rather we send them to a device for execution with the necessary data having already been set. The process of setting kernel arguments is discussed in Section 4.3.4 on page 44.

4.3.1 Kernel Header

Information which we need to acquire from a kernel header includes its name and return type. To acquire these details, we parse the kernel header, using the kernel file as a string. Once we have the kernel name and return type it is stored in the program

structure which is illustrated in Figure 4.2. The kernel as a string is also stored in the program structure as source code.

```
struct _cl_program
{
        callable_kernel procedure;
        char * return_type;
        char * kernel_name;
        struct _cl_program_parameter * parameters;
        char * source_code;
        cl_int num_parameters;
        char * file_name;
};
```

Figure 4.2: Program Type Definition

The process of parsing the kernel header involves establishing where the kernel identifier is situated in the kernel string source. Once we have acquire this, we know that the next string after a white-space will be the return type. This is saved in a character array named `return_type` in the program. Once we know where out return type ends, we can acquire the kernels name by finding where the parameter list starts. This is because the kernel name is situated between the return type and the parameter list. To get the kernel name, we look for the opening parentheses of the parameter list. Now we can copy the string between the return type and the parameter list. We place the copy of this string in a character array named `kernel_name`.

4.3.2 Kernel Parameters

Extracting a kernel's parameters is done using a similar manner to that of getting a kernel name and return type details. We parse the source code between the parentheses of the kernel parameter list using the kernel source string. Details which we look for include the visibility of the parameter, the parameter type and the parameter's name. These are stored in the program structure which is illustrated in Figure 4.3. This structure is allocated memory from the primary memory of the system.

Before we actually get the parameter details, we count how many parameters a kernel has by parsing the kernel source string. We discovered that we need to know in advance how many arguments a kernel can take. Not knowing how many arguments a kernel could resulted in a very implementation of the function to set an argument. For this reason we count the number of parameters. This count is stored in the program structure as an integer type named `num_parameters`. It is important to know how many parameters we need to store, because we implement a parameter structure which

only contains the details on a single parameter. This results in us storing parameter details in an array. This method of storing parameters in an array was necessitated by the need to store multiple details of each parameter.

```
struct _cl_program_parameter
{
        char * arg_vis;
        char * arg_type;
        char * arg_name;
};
```

Figure 4.3: Kernel Parameter Type Definition

The process of parsing the kernel source string involves identifying where the opening parentheses and closing parentheses of the parameter list are situated. Once we have found these two points, we create a temporary pointer which is initially set to the opening parenthesis. The parameters are divided by commas in the source string. Using this we divide the parsing process into parsing a parameter at a time. Another temporary pointer is used to point to the end of a parameter,but it is initially set to the position just after the starting point. We iterate the second temporary pointer until we find either a comma or reach the end of the list. We know that between the two temporary points, we have a parameter. From this we look to see if it has a visibility, which is listed first in a parameter. If it does have a visibility specified we store the visibility in the character array named `arg_vis`, else we set the visibility to null. The last value listed in a parameter is its name, to acquire the name we work from the end of the parameter back to the first white-space. The name is stored in a character array named `arg_name`. Anything between the visibility and the name is considered to be the parameter type, this is stored in a character array named `arg_type`.

The parameter visibility refers to where the parameter resides in memory. Parameters can be declared as global, shared or local. It is also possible for a parameter not to have these definitions, in this case a parameter is used to pass a set value to the kernel and does not get changed by the kernel during execution. Because of this, there is no need to specify the access allowed to the data stored at these locations.

The type is stored so that we can specify how to pass the arguments to the kernel from the calling function. This is dealt with in greater detail in Section 3.2.3.1 on page 33.

4.3.3 Building a Program

To be able to run kernels, we make use of a temporary file. To create a temporary file, we use the kernel source code, its header details and its parameter list details. Inside

the temporary file we place the kernel source code and create the calling function. Once this has been completed we compile the temporary file.

Creating the kernel calling function involves setting its name to the kernel's name and creating a parameter list. The parameter takes in the kernel's global identifier, a list of kernel parameters and the return type. In the calling function, we call set the kernel global identifier using the global identifier argument. We also insert a call to the kernel using the kernel name and the kernel arguments.

The manner in which the arguments are passed to the kernel depends on its type. There are two cases which we deal with, a parameter which has been specified with a kernel visibility and a parameter which has been specified without a kernel visibility. When an argument has a visibility specified, then it means that the argument is a buffer. For this reason we need to type cast it as our implemented memory type. To access the data stored in the memory type we need to de-reference the data which is cast to a void pointer. In the case where a parameter has no specified visibility, we cast it as its type, discussed in Subsection 4.3.2 on page 42, and de-reference it to access the parameter. These are a rather complicated methods of passing arguments, but we had issues trying to access the arguments. We found that this was one method which allowed us to pass the correct argument value properly. Future work could involve attempting to simplify the manner in which we pass the arguments to the kernel.

Once the temporary file has been completed, we compile it using the C compiler. Then we get a handle on the compiled temporary file so that we can call it a execution time. The file handle is stored in the program structure. To get a handle on the file, it needs to be in the same directory as the source code. We discovered that if the temporary file is not created in the same directory as the source code, then we cannot access the file after it has been compiled. To get this working, we specified that the file be created in the current directory.

4.3.4 Setting Kernel Arguments

Because the data passed to kernels as arguments does not differ from one kernel to another, kernel arguments are set at run-time before execution takes place. When an argument is set, it is stored in an argument structure as illustrated in Figure 4.4. Similar to a kernel's parameter array mentioned in Subsection 4.3.2 on page 42, there is also an array of arguments. This array is pointed to by the a kernel argument list pointer. A kernel type is illustrated in Figure 4.5.

```
struct _cl_kernel_argument
{
        size_t arg_size;
        void * arg_value;
};
```

Figure 4.4: Kernel Argument Type Definition

Storing arguments depends on an arguments value. There are two distinct cases relating to an argument's value, the first case is when an argument has a non-null value and the second is when it has a null value. In the first case, the argument has already been declared. Because the argument already exists, we also know its size. Using the size we can allocate memory to store it in the kernel argument type using the void pointer named `arg_value`. Following this, we can copy the arguments value from memory and store it in the allocated memory. The second case deals with arguments with values which have been specified as null. Within this case, there are two further cases. The first being that the argument size is not set to zero and the second having the argument size set to zero.

```
struct _cl_kernel
{
        cl_program      program;
    const char *    kernel_name;
        struct _cl_kernel_argument * arguments;
        cl_int num_arguments;
        cl_int global_id;
};
```

Figure 4.5: Kernel Type Definition

In the case where the argument size is not set to zero, we allocate a memory block using the specified size. We then allocate a new memory type and set its pointer to point to the memory block. Finally, the memory type is stored as the argument value in the argument type. This case is essentially creating a local buffer for use by a kernel. When the argument size is set to zero, then we set the argument value in the argument type to null.

4.4 Execution Model

The main focus of implementing the execution model involves ensuring that OpenCL programs are executed in parallel. To ensure that we can execute programs, we make

use of command queues to hold commands which need to be executed. Parallelism is provided by the use of the OpenMP API, which provides us with threading capabilities [3].

Languages like C do not have any native methods which provide command queues, we need to provide one ourselves. The implementation of the command queue is discussed in Subsection 4.4.1.

Another aspect of the execution model is the use of events to specify command types and their execution status. Although OpenCL programs can be written to make use of events, they are not required to ensure that a queued command is executed. We take this into account when implementing our command queue. Events are discussed in Subsection 4.4.2.

In our implementation there are two methods with which commands can be executed. The method by which a command is executed is determined by whether an event is specified or not. This is discussed in Subsection 4.4.3.

4.4.1 Command Queue

We design a command queue to hold a list of queue items. This allows the the command queue to point to a dynamic array of queue items. If a command item is en-queued and the number of en-queued commands plus the command waiting to be en-queued exceeds the size of the queue, then the queue is re-sized. This is performed by creating a new array which is one item larger than the old array. We copy the old array's data into the new array and free up memory associated with the old array. The one major disadvantage of this method is that it is inefficient. Future extensions to the project could include improving the implementation of the command queue to be more efficient.

```
struct _cl_queue
{
        void * command;
        cl_command_type command_type;
};
```

Figure 4.6: Command Queue Item

Every OpenCL device is intended to have a command queue associated with it. Contrary to this, we do not create a command queue which is associated with the CPU, we rather create a command queue which is only used to store commands to be executed. Reasons for this are twofold; we do not possess any method for creating command

queues associated with a CPU and our command queues do not actually send any commands to the processor to tell it to execute a command. Our command queue rather acts as a holder for commands which are accessed at execution time in a manner depending on the methods mentioned in Section 4.4 above.

```
struct _cl_buffer
{
        void * dest;
        void * source;
        size_t size;
};
```

Figure 4.7: En-queued Buffer Type

As mentioned previously, a command queue contains a list of queue items. A queue item is defined in Figure 4.6. Each queue item holds the command to be executed and the command type. This allows us to execute commands without the need for an event object. There are three command types which we catered for in our implementation; reading from a buffer, writing to a buffer and executing kernels. Because of this we make use of the kernel structure defined in Figure 4.5 on page 45 and the buffer structure defined in Figure 4.7 respectively. These structures are set as the command of a queue object, while we specify which command type the queue object contains. This allows us to determine how to execute a command at execution time. Setting the command of a queue item can be separated into two distinct cases, setting a kernel for execution and setting a buffer to be read from or written to.

In the case of a kernel being the command to be en-queued, the kernel structure is set as the command of the queue item. In the case of a buffer read or write command being en-queued; the source data, destination pointer and size of the data are stored in a buffer structure illustrated in Figure 4.7. This is then set as the command of the queue item.

4.4.2 Events

An event is associated with a command which is en-queued in a command queue. Each event points to a command queue in which its associated command is en-queued. We also require that we know what type of command it is, its execution status and its queue number in the command queue. The queue number allows us to keep track of the position of a command in a command queue. Knowing the position allows us to perform out-of-order execution of commands. An event is defined in Figure 4.8.

```
struct _cl_event
{
        struct _cl_command_queue * command_queue;
        cl_command_type command_type;
        cl_int status;
        cl_int event_queue_number;
};
```

Figure 4.8: Event Type Definition

4.4.3 Program Execution

As mentioned previously in Subsection 4.4.2 on the preceding page, if an event is specified, then we can perform out-of-order execution of commands. It is also possible not to use events for execution in a program. The result of this is in-order execution of commands from the command queue. In the case where no events are used, then all commands are en-queued before execution time. Once the commands have been en-queued, then they are all executed sequentially by looping through the command queue's queue items.

In the case where events are used in a program, there are two options available. The first was stated in the previous section, where one command is queued at time and then that command is executed before another command is en-queued. The second option is when all the commands are en-queued as mentioned earlier, however commands can be executed out-of-order by using the associated event.

The actual execution of a program is similar in both the methods mentioned above. We initially check to see what command needs to be executed, this can be achieved by using the command queue or event. This again depends on the method used. Next we extract the command and execute depending on its type. When we execute a kernel we use the OpenMP API [3] to ensure that out kernels run in parallel on the CPU using threading. When reading data from a buffer, we perform a memory copy of the the buffer and assign the data to the destination. When writing data to a buffer, we perform a memory copy of the data source and assign the data to the buffer.

The process of executing a kernel involves running a loop of the number of global threads used for kernels. However, executing kernels in such a manner results in sequential execution of kernels. To achieve parallel execution of kernels, we make use of the OpenMP API to provide threading capabilities without the need to manage threads ourselves. This is achieved by parallelizing the for loop using OpenMP pragmas to specify a parallel section which incorporates a for-loop. We set the number of threads equal to the number of processing cores available on a system. By using such

a method we can scale our number of kernels being executed at a time equal to the number of processing cores we have.

At execution time, the kernel calling function is passed a global identifier, which it then assigns to a variable which is private to each instance of a kernel. This global identifier allows each kernel to identify which set of the data it needs to use to compute its result. Other variables which can be used include a local identifier to identify its position in a group and a group identifier to identify the group a kernel belongs to. We chose not to implement these two as they are not always used. If they are required in future, then out implementation could be adapted to use them rather easily.

4.5 Implementation Summary

In this chapter we have described how we implemented OpenCL for a CPU parallel architecture. We implement OpenCL as a library of functions rather than a compiler. The four main aspects which need to be implemented are: devices and context, buffers, kernels and the execution model. Our implementation of these aspects sometimes does not utilize them as intended. The main reason for this is that our implementation is initially aimed at making use of a single multi-core CPU in a system. To effect parallel processing in our system, we make use of the OpenMP API.

Chapter 5

Evaluation

In this section we evaluate the performance of our implementation based on three main criteria, these being: parallel execution of a program on the CPU versus sequential execution of the same program on the CPU, execution of a program in parallel on the CPU compared to the performance of the same program executed on a GPU implementation by NVIDIA and evaluating the performance of our memory implementation versus the performance of the GPU implementation.

Experiments were performed on an Intel Core2 Quad Q9400 running at 2.66GHz with 4GB of main memory. Our operating system used is Fedora Core 10. Experiments using GPUs were performed using a NVIDIA GTX275 graphics card with 896MB of memory. We used NVIDIA's OpenCL 1.0 Conformant Release.

5.1 Parallel versus Sequential Performance

5.1.1 Purpose

The purpose of this experiment of the system is to evaluate the parallel performance of our CPU implementation with the standard sequential performance also using a CPU. We choose a sample program which is used to compute a Mandelbrot fractal. Computation of Mandelbrot fractals are very parallel in nature and are thus a good method to test the implementation of the parallel execution model of our OpenCL implementation.

5.1.2 Process

Evaluating the performance of our parallel OpenCL implementation for the CPU compared with the performance of sequential execution on a CPU, involves running the

same program both in parallel and sequentially. This process involves calculating a Mandelbrot fractal ten times using a set width and height. These calculations are performed both in parallel and sequentially. For the purpose of this evaluation we compare the two samples, with each sample consisting of ten runs.

For the calculation, we set the width and height of the Mandelbrot fractal to 8192, the maximum number of iterations per point is 20 and the scale is set to three. Our number of threads used to execute kernels is set to four.

We time each of the parallel and sequential executions per run in an attempt to ascertain how the performance of the two execution methods differ. The timings made for the parallel executions include all run time operations, these include the setting of kernel arguments, creation of a kernel, en-queuing and execution of the kernel, and en-queuing and execution of a read buffer command. The timings made for the sequential execution include the call to the function to calculate the fractal.

5.1.3 Precautions

To ensure that our results are as consistent as possible, we have as few programs running as possible at the time of testing. Tests are conducted on the same machine using the same system settings for each run. We also ensure that no unnecessary input/output operations take place during our timing sequence.

5.1.4 Results

Results of parallel versus sequential execution times are displayed in Figure 5.1 on the next page. Parallel execution times are represented by a red line, while sequential execution times are represented by a blue line. The values on the x-axis are the run numbers. The y-axis values are the times taken for an execution to complete for each run, the time is displayed in seconds.

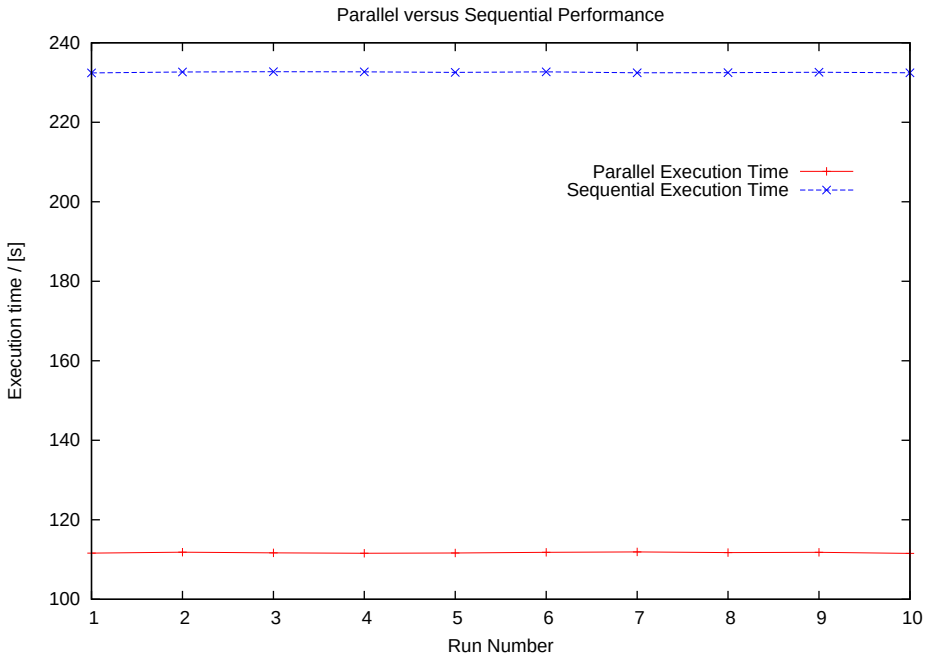

Figure 5.1: Parallel versus Sequential Performance

5.1.5 Analysis

Looking at the results in Figure 5.1 and analysis in Table 5.1, we notice that there variance in the parallel execution times. A closer inspection of the standard deviation of the parallel execution times reveals that there is a deviation of 0.117 seconds over the sample. There is inconclusive evidence into what causes this, there are several possibilities which can affect the times. Some of these are the amount of work, the number of threads being used or the operating system itself could affect the timings. In an attempt to find out what the cause may be, we experiment further by assessing what effect the amount of work has on the timing variance. This experiment can be found at Section 5.2 on the following page. Future work could involve further testing in an attempt to establish which cause has the greatest effect, if at all.

Execution Type	Standard Deviation
Parallel	0.117
Sequential	0.114

Table 5.1: CPU versus GPU Results Analysis

The purpose of this experiment was an attempt to establish whether our parallel implementation for the CPU is better than a sequential version also running on the CPU.

Initial results indicate that our parallel implementation has a lower execution time than a sequential version.

5.2 Effect of the Amount of Work on Timing Variance

5.2.1 Purpose

The purpose of this experimentation is to establish the effect that the amount of work has on timing variance. This experiment follows on from the analysis of the results made in Section 5.1. For this experiment we choose a sample program which is used to calculate Mandelbrot fractals. To vary the amount of work, we vary the height and width of the fractal.

5.2.2 Process

The process of evaluating the effect that the amount of work has on the timing of a sample, involves recording execution times of a sample program whilst varying the amount of work for each sample. Each sample is executed ten times with a fixed size. Each execution is considered to be a run, with every tens runs constituting a pass. Each pass will use a different width and height.

The fractal width and height values are varied from 128 to 8192, increasing in powers of two. To ensure that our results reflect the effect of the amount of work, we keep the maximum number of iterations per point constant at 20 and the scale remains constant at three. We also ensure that each run is executed using four threads. The width values for each pass are listed in Table 5.2.

Pass	**1**	**2**	**3**	**4**	**5**	**6**
Width	128	256	512	1024	2048	4096

Table 5.2: Width of fractal per Pass

We time each execution to evaluate what effect the amount of work has on the performance. The timings made include all run time operations, these include the setting of kernel arguments, creation of a kernel, en-queuing and execution of the kernel, and en-queuing and execution of a read buffer command. Once we have recorded all of our timings, we calculate the standard deviation of the population of each pass.

5.2.3 Precautions

To ensure that our results are as consistent as possible, we have as few programs running as possible at the time of testing. Tests are conducted on the same machine using the same system settings for each run. Each of the passes, including ten runs per pass, are conducted one after another using a loops in the code. We also ensure that no unnecessary input/output operations take place during our timing sequence.

The program's values, apart from the width and height are kept constant to ensure that the results we obtain reflect how the performance relates to the amount of work rather than how the performance relates to the number of threads being used for execution.

5.2.4 Results

Results of the effect of amount of work on timings are displayed in Figure 5.2. The standard deviation of the population of execution times in relation to fractal width are represented by a red line. The values on the x-axis are related to the width of the Mandelbrot fractal. The y-axis values are the standard deviation of the population of execution times in seconds.

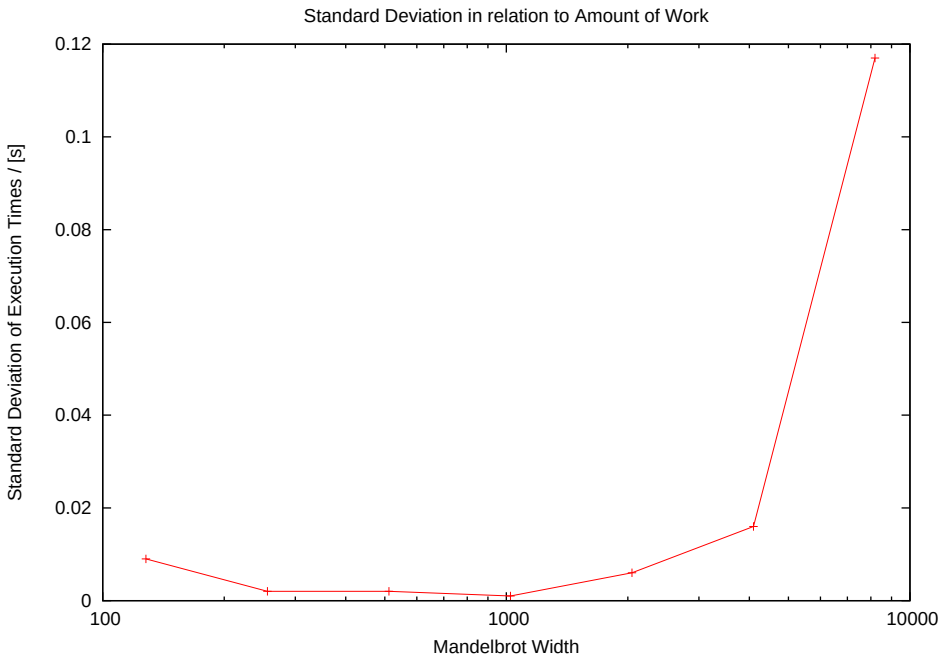

Figure 5.2: CPU Performance based Work

5.2.5 Analysis

Looking at the results in Figure 5.2 on the preceding page, we notice that the initial value, using a width of 128, has a slightly higher standard deviation than its neighbours. This could be an effect of the operating system, however further testing would need to be carried out to satisfy the actual reason. If we were to factor out the first time recorded for a width of 128, then we would have a standard deviation of 0.001 seconds, which is significantly lower than the value in Table Table 5.3 on page 55 and more in-line with the other values.

Width	Standard Deviation
128	0.009
256	0.002
512	0.002
1024	0.001
2048	0.006
4096	0.016
8192	0.117

Table 5.3: Standard Deviation of Samples

Increasing the width of the fractal, we notice that for widths of 256, 512, 1024 and 2048 there is very little difference between the sample times. However, from a width of 1024 there is an initial gradual increase until the width reaches a value of 4096. At this point, we notice that there is a large increase in the variance when the width approaches 8192.

From our analysis, there is a trend of the variance in the times for a pass increasing as the amount of work (width) increases.

5.3 Threading Performance

5.3.1 Purpose

This experiment attempts to evaluate the effect of increasing the number of threads used for executing programs. We use a program to calculate Mandelbrot fractals with a fixed size and width. We choose to use a sample program which calculates Mandelbrot fractals since the size of the fractal being calculated allows for accurate timings to be taken.

5.3.2 Process

The process of evaluating the effect of the number of threads used for execution on the performance, involves varying the number of threads used to calculate a program. This process involves executing the Mandelbrot fractal program ten times for a set number of threads. These ten executions constitute a single pass. The size of the fractal calculated has a width of 1024 and a height of 1024, the maximum number of iterations per point is set to 20 and the scale is set to three. To enable us to compare the effect of the threads, we vary the threads in powers of two starting with two threads and increasing to 256 threads. The actual number of threads used for each pass is displayed in Table 5.4. The number of processing cores remains constant at four throughout the experiment.

Pass	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Number of Threads	2	4	8	16	32	64	128	256

Table 5.4: Number of Threads used

We time each execution to evaluate how the number of threads affect the performance. The timings made include all run time operations, these include the setting of kernel arguments, creation of a kernel, en-queuing and execution of the kernel, and en-queuing and execution of a read buffer command. Once we have recorded all of our timings for a single pass, we calculate the average execution time of the pass for each set number of threads.

5.3.3 Precautions

To ensure that our results are as consistent as possible, we have as few programs running as possible at the time of testing. Tests are conducted on the same machine using the same system settings for each run. All ten runs are conducted one after another using a loop in the code. Our CPU implementation necessitates that we hard code the number of threads used for each execution pass. We also ensure that no unnecessary input/output operations take place during our timing sequence.We use the average times for each pass in an attempt to even ensure that any variance in our timings does not affect our analysis of the timings.

The program's values are kept constant to ensure that the results we obtain reflect how the performance relates to the number of threads rather than how the performance relates to the amount of work being done.

5.3.4 Results

Results on the effect of the number of threads used for execution are displayed in Figure 5.3. Execution times are represented by a red line. The values on the x-axis are the number of threads used. Values on the y-axis are the execution times in seconds.

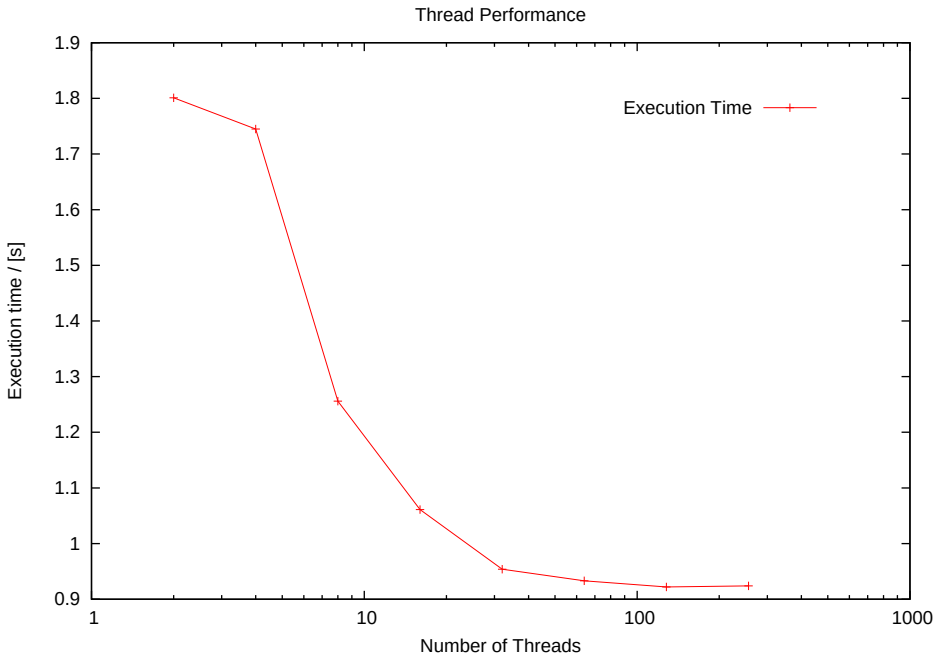

Figure 5.3: Thread Performance

5.3.5 Analysis

Looking at the results in Figure 5.3, we notice that as the number of threads increase, the execution times decrease. However, the decrease in time between two threads and four threads is not as large as the difference between four threads and 32 threads. However increasing the number of threads beyond 32 threads starts to yield diminishing returns. The standard deviation in execution times between 128 threads and 256 threads is approaching zero. Standard Deviation between two threads are displayed in Table Table 5.5 on page 58.

Threads	Standard Deviation
2-4	0.028
4-8	0.245
8-16	0.098
16-32	0.053
32-64	0.011
64-128	0.006
128-256	0.001

Table 5.5: Standard Deviation between Number of Threads

From our analysis we find that increasing the number of threads initially yields large decreases in execution times. However when approaching 32 threads, used by four processing cores, we find that any further increase in the number of threads starts to yield diminishing returns. The point at which execution times fail to decrease any further is at a time of around 0.92 seconds, with the number of threads increasing towards 256.

Future work could involve finding our how this effect scales to increasing the number of processor cores used for execution.

5.4 CPU versus GPU Performance

5.4.1 Purpose

The purpose of this experiment is to evaluate the performance of our implementation of OpenCL for CPUs with the implementation of OpenCL for GPUs created by NVIDIA. We choose a program which multiplies an array of integer numbers by two. This decision was made as it not only tests the execution times, but also allows us to evaluate our implementation against one which has been created for mainstream use.

5.4.2 Process

The process of evaluating the performance of our implementation for the CPU versus a GPU implementation, involves executing the same program the same number of times on the different device types. The chosen program, used to calculate the multiples of an array of numbers is executed ten times on each device type. Furthermore, executing a program ten times constitutes a single pass, in an attempt to discover any interesting features of the implementations, we make multiple passes varying the size of the array. This affects how much work is done by each kernel.

Each execution is timed in an attempt to evaluate the performance of the implementation for each device. The timings made include all run time operations, these include the setting of kernel arguments, creation of a kernel, en-queuing and execution of the kernel, and en-queuing and execution of a read buffer command. Once we have recorded all of our timings for a single pass, we calculate the mean execution time of the pass for each device.

To change the amount of work done in each pass, we vary the size of the array we are using to calculate each number multiplied by two. We start off calculating the multiples of 10000000 numbers. For each pass, we increase the array size until we have an array size of 10000000. The amount of numbers in an array which are calculated per pass are displayed in Table 5.6.

Pass	**1**	**2**	**3**	**4**	**5**	**6**
Size of Buffer	1000000	2000000	4000000	6000000	8000000	10000000

Table 5.6: Numbers in Array Calculated per Pass

It must be noted that this experiment favours the CPU implementation more, as we do not optimize the number of work groups and the number of work items contained in the work groups for use on the GPU. We believe that further optimizations could improve the GPU results further. Future work could involve trying to optimize the experiment for the GPU.

5.4.3 Precautions

To ensure that our results are as consistent as possible, we have as few programs running as possible at the time of testing. Tests are conducted on the same machine using the same system settings for each run. All of our passes and each of their ten runs are conducted one after another using loops in the code. We also ensure that no unnecessary input/output operations take place during our timing sequence. We use the average times for each pass in an attempt to even ensure that any variance in our timings does not affect our analysis of the timings.

We make use of the same test program with as few adjustments as possible to ensure that we attempt to record results which can be compared fairly with one another. Adjustments made to the code for the NVIDIA implementation include adding the NVIDIA OpenCL library and adjusting the work groups and the number of work items per work groups. Our CPU implementation only makes use of one work item per work group, but the NVIDIA implementation is limited to a maximum number of work groups which can be used.

5.4.4 Results

Results on CPU performance in comparison to GPU performance are displayed in Figure 5.4. The CPU execution times are are represented by a red line, while the GPU execution times are represented by a blue line. The values on the x-axis are the size of the array, which contains integer numbers. The values on the y-axis are the execution times in seconds.

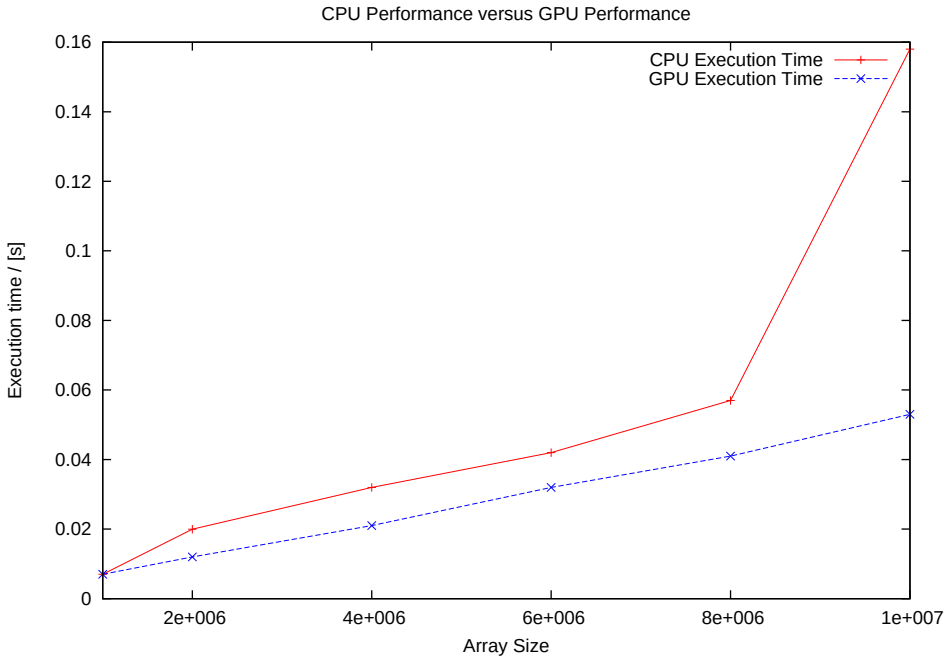

Figure 5.4: CPU Performance versus GPU Performance

5.4.5 Analysis

Looking at the results in Figure 5.4, we notice that the GPU performance is very linear throughout all six passes. However, the CPU performance is a lot less linear. Initially, the between array sizes of 1000000 to 8000000, the curve is relatively linear. However as we increase the array size to 10000000, we notice that the CPU times increase drastically. Between array sizes of 1000000 and 8000000, our CPU implementation has execution times which are relatively close to those of the GPU implementation. This trend is indicated by the difference between GPU and CPU execution times in Table Table 5.7 on page 61.

Width	Difference
100000	0.001
2000000	0.007
4000000	0.011
6000000	0.010
8000000	0.015
10000000	0.105

Table 5.7: Difference in execution times

From our initial analysis, it is obvious that the GPU performance scales very well to an increase in the size of the array to be calculated. Conversely, our CPU implementation's performance does not scale very well to an increase in array size. The results achieved are comparable to the results achieved by Stratton et al. [14], where they found that the performance of MCUDA on a CPU is lower than the performance of CUDA on the GPU. However, our GPU implementation has not been optimized to take advantage of the architecture. For this reason we cannot state that the performance of the CPU implementation is significantly lower than the performance of the GPU implementation.

Future work could involve performing a comparison test between the two implementations which are more computationally intensive. Something such as calculating large Mandelbrot fractals could be a potential test. Other options are tests which were conducted by Stratton et al. [14], namely matrix multiplication, Coulombic Potential and high-resolution MRI image reconstruction. Using these test could involve testing both the CPU and GPU implementations against the baseline CUDA implementation for the GPU.

5.5 Memory Performance

5.5.1 Purpose

The purpose of this experiment is to evaluate how our implementation of buffers in primary memory performs in comparison to the performance of buffers in NVIDIA's GPU implementation. To achieve this we perform both reads and writes using buffers for both implementation. The size of a buffer being read or written to is varied in an attempt to establish what effect different buffers sizes have on performance.

5.5.2 Process

The process of evaluating memory performance can be divided into read performance and write performance. This performance testing is divided into two separate parts;

with the first part testing the read performance of our CPU implementation in comparison to the read performance of NVIDIA's GPU implementation. The second part tests the write performance of our CPU implementation in comparison to the write performance of NVIDIA's GPU implementation.

For testing purposes we use a buffer which can contains 2560000 integer numbers. We initially write to the buffer and record the time taken to write a set amount of data to a buffer ten times. This constitutes a single pass. The same is done for reading data from a buffer, except that we record the time taken to read a set amount of data from a buffer ten times. The timings made include the en-queuing of a write or read command and the execution of the write or read buffer command. Once we have recorded all of our timings for a single pass, we calculate the mean execution time of the pass for each device.

We vary the size of memory which is either being read from or written to for each pass. Our initial buffer size is set to 2500 integer numbers, the buffer size is incremented in multiples of four, using the starting size as our base multiple value, until it reaches a size of 2560000 integer numbers. The actual values for each pass are displayed in Table 5.8.

Pass	**1**	**2**	**3**	**4**	**5**	**6**
Size of Buffer	2500	10000	40000	160000	640000	2560000

Table 5.8: Buffer Sizes

The number of kernels, also referred to as global threads, used in the execution depend on the size of the array, also referred to as the width. For our CPU implementation, this is set in a one-to-one relationship, in effect $global\,threads = width$. However, the GPU implementation does not support the large array sizes. For this reason, the number global threads used in the execution by the GPU implementation is set to $width/20$ and the number of local threads is set to $local\,threads = 20$.

5.5.3 Precautions

To ensure that our results are as consistent as possible, we have as few programs running as possible at the time of testing. Tests are conducted on the same machine using the same system settings for each run. Each of the ten runs per pass are conducted one after another using a loop in the code. We also ensure that no unnecessary input/output operations take place during our timing sequence.

Execution times for reading from a buffer and writing to a buffer are recorded separately to ensure that one memory operation does not have an effect on the other. All ten

passes are completed together using loops in the code. We use the average times for each pass in an attempt to even ensure that any variance in our timings does not affect our analysis of the timings.

Adjustments made to the code for the NVIDIA implementation include adding the NVIDIA OpenCL library and adjusting the work groups and the number of work items per work groups.

5.5.4 Results

Results of memory read performance is displayed in Figure 5.5, while results of memory write performance is displayed in Figure 5.6 on the following page. In both figures, CPU memory times are represented by red lines. GPU memory times are represented by blue lines. The x-axis values are the size of the array, which contains integer numbers. The y-axis values are the execution times in seconds.

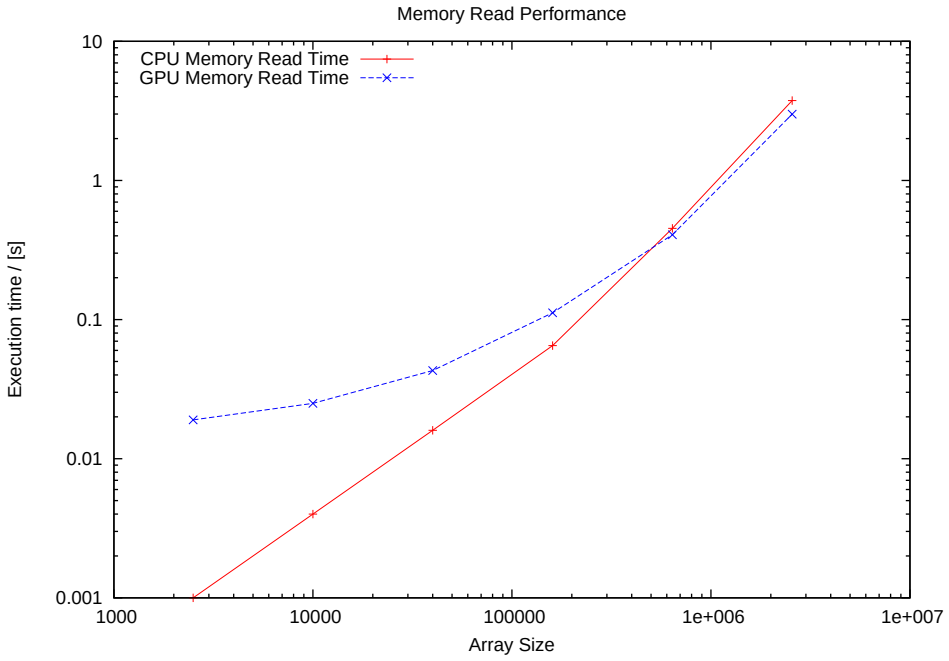

Figure 5.5: Memory Read Performance

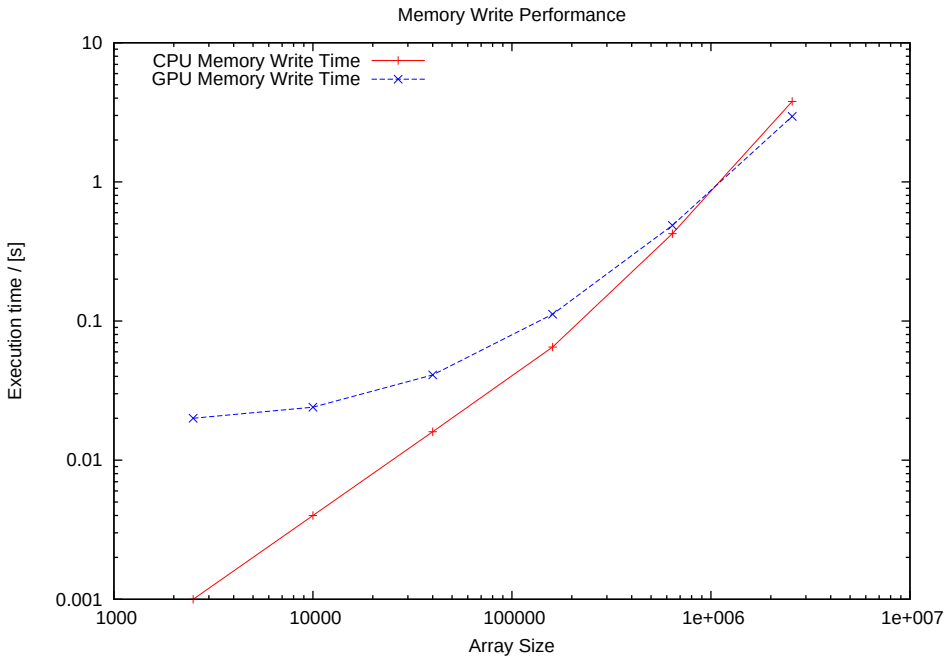

Figure 5.6: Memory Write Performance

5.5.5 Analysis

From the results displayed in Figure 5.5 on the previous page and Figure 5.6, we find that both the read performance and write performance are very similar. Focusing on the CPU memory read and write performance, we observe that the performance is relatively linear. In comparison to this, GPU memory read and write performance is far less linear.

The GPU memory performance has larger increases in read and write times as the array size grows. This could be related to the various memory types available on a GPU device. The reason for CPU read and write performance being close to being linear could be caused by the manner in which our buffers are implemented using primary memory. Primary memory has a relatively uniform access speed which is linked to the CPU clock-speed.

5.6 Evaluation Summary

In this section we have attempted to evaluate the performance of our implementation using a number of different experiments. Experiments we performed were:

- performance of parallel execution versus sequential execution
- the effect that the amount of work has on timing variance
- the effect that the number of threads has on timings
- the performance of our CPU implementation versus NVIDIA's GPU implementation
- The performance of our memory implementation compared with the memory implemented by NVIDIA

We analyzed the resulting data and attempted to draw some conclusions about the data. Final conclusions on the results are in Chapter 6 on the next page.

Chapter 6

Conclusion

6.1 Summary

In this project we have attempted to develop an approach for implementing OpenCL for multi-core CPUs. In our attempt to develop an approach we: reviewed work which relates to our problem, assessed the design of OpenCL, explained how we implemented the design and performed an evaluation of the system to test our implementation.

In our reviewed work, we discovered that areas which relate to our problem include the use of threading to effect parallel processing, heterogeneous computing and translating languages from one architecture to another.

In our design section we analyzed the differences in mapping OpenCL and its components different parallel architecture. The parallel architectures which we considered include a GPU parallel architecture and a CPU parallel architecture.

In our implementation section we described how we implemented OpenCL components to make use of the CPU parallel architecture. The components which we described include: devices and context, buffers, kernels and the execution model. In describing the implementation process, we discussed issues we encountered and how we lose functionality by implementing the components in the chosen method.

Our evaluation section attempts to establish how well our implementation performs. This process involved testing our implementation using five different experiments. The experiments conducted included: comparing parallel and sequential execution, analyzing the effect of the amount of work on timing variance, threading performance, CPU versus GPU performance and memory performance.

6.2 Conclusions

The conclusions we make with regards to implementing OpenCL for multi-core CPUs are:

1. By analyzing how the design of OpenCL maps to the GPU parallel architecture and the CPU parallel architecture, we discover that OpenCL is more suited for use with a GPU parallel architecture.

2. It is possible to implement OpenCL for multi-core CPUs. We achieved this by creating a library of OpenCL function calls using C code. By implementing OpenCL in this manner, we discovered that we lose functionality of the OpenCL system.

3. Our parallel implementation of OpenCL has better performance than the same program being executed sequentially.

4. Executing programs using our implementation results in variance in execution times. Leading on from this, we conclude that our implementation is not efficient.

5. The amount of work done has an effect on the variance in execution times. The effect we uncover is that larger amounts of work result in more variance in execution times.

6. Increasing the number of threads used for execution, decreases execution times of kernels. However, as the number of threads increase, the execution times start to yield diminishing returns.

7. The performance of our CPU implementation is lower than the performance of NVIDIA's GPU implementation. Furthermore, these results indicate that GPUs are better suited for parallel processing.

8. The performance of our memory implementation is linear, compared to this, the GPU memory implementation is less efficient than our implementation.

References

[1] Sarita V. Adve, Vikram S. Adve, Gul Agha, Matthew I. Frank, María Jesús Garzarán, John C. Hart, Wen-mei W. Hwu, Ralph E. Johnson, Laxmikant V. Kale, Rakesh Kumar, Marinov Darko, Klara Nahrstedt, David Padua, Madhusudan Parthasarathy, Sanjay J. Patel, Rosu. Grigore, Dan Roth, Marc Snir, Josep Torrellas, and Craig Zilles. Parallel computing research at Illinois the UPCRC agenda. Technical report, 201 N Goodwin Ave, Urbana, IL 61801-2302, 2008.

[2] AMD. *Brook+*. Advanced Micro Devices, Novemeber 2007.

[3] OpenMP Architecture Review Board. *OpenMP Application Program Interface*. OpenMP Architecture Review Board, May 2008.

[4] Sha'Kia Boggan and Daniel M. Pressel. GPUs an emerging platform for general-purpose computation. Technical report, US Army Research Laboratory, August 2007.

[5] Khronos OpenCL Working Group. *The OpenCL Specification*. Khronos Working Group, 2009.

[6] Tom R. Halfhill. Parallel processing with CUDA. `http://www.MPRonline.com`, January 2008.

[7] Byunghyun Jang, Synho Do, Homer Pien, and David Kaeli. Architecture-aware optimization targeting multithreaded stream computing. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 62–70, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-517-8. doi: http://doi.acm.org/10.1145/1513895.1513903.

[8] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 64, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2143-6.

[9] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 101–110, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-397-6. doi: http://doi.acm.org/10.1145/1504176.1504194.

[10] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Meng. Merge: a programming model for heterogeneous multi-core systems. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 287–296, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-958-6. doi: http://doi.acm.org/10.1145/1346281.1346318.

[11] NVIDIA. *NVIDIA CUDA Programming Guide 2.0*. NVIDIA Corporation, 2008.

[12] 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 *PPoPP '08 Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-795-7. doi: httpdoi.acm.org10.11451345206.1345220.

[13] 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 *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 195–204, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-978-4. doi: http://doi.acm.org/10.1145/1356058.1356084.

[14] John Stratton, Sam Stone, and Wen mei Hwu. MCUDA: An efficient implementation of CUDA kernels for multi-core cpus. In *21st Annual Workshop on Languages and Compilers for Parallel Computing (LCPC'2008)*, July 2008. URL `http://www.gigascale.org/pubs/1328.html`.

[15] Perry H. Wang, Jamison D. Collins, Gautham N. Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y. Yang, Guei-Yuan Lueh, and Hong Wang. Exochi: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 156–166, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi: http://doi.acm.org/10.1145/1250734.1250753.

Appendix A

Experiment Data

Execution	1	2	3	4	5	6	7	8	9	10
Parallel	111.627	111.859	111.672	111.583	111.640	111.797	111.906	111.749	111.822	111.547
Sequential	232.404	232.666	232.176	232.691	232.545	232.699	232.455	232.492	232.569	232.437

Table A.1: Parallel versus Sequential Performance

Width	1	2	3	4	5	6	7	8	9	10
128	0.060	0.028	0.028	0.028	0.028	0.032	0.03	0.030	0.030	0.030
256	0.119	0.112	0.112	0.112	0.112	0.112	0.112	0.112	0.112	0.112
512	0.452	0.444	0.445	0.445	0.444	0.445	0.445	0.444	0.444	0.445
1024	1.779	1.778	1.779	1.776	1.777	1.779	1.778	1.775	1.778	1.779
2048	7.088	7.072	7.070	7.070	7.068	7.078	7.071	7.070	7.070	7.073
4096	28.292	28.258	28.240	28.248	28.239	28.268	28.245	28.250	28.251	28.273
8192	111.627	111.859	111.672	111.583	111.640	111.797	111.906	111.749	111.822	111.547

Table A.2: Effect of the Amount of Work on Timing Variance

Threads	1	2	3	4	5	6	7	8	9	10
2	1.801	1.801	1.800	1.800	1.801	1.801	1.801	1.801	1.801	1.800
4	1.757	1.743	1.744	1.743	1.743	1.746	1.743	1.743	1.743	1.746
8	1.226	1.259	1.259	1.236	1.244	1.265	1.264	1.268	1.269	1.264
16	1.223	0.941	1.080	1.115	1.114	1.068	1.036	1.132	0.937	0.962
32	1.082	0.948	0.944	0.962	0.947	0.919	0.931	0.934	0.947	0.928
64	1.034	0.931	0.905	0.923	0.915	0.946	0.909	0.910	0.913	0.945
128	0.961	0.914	0.914	0.924	0.925	0.925	0.915	0.921	0.908	0.908
256	0.941	0.912	0.976	0.921	0.921	0.914	0.904	0.923	0.914	0.911

Table A.3: Thread Performance

Array Size	1	2	3	4	5	6	7	8	9	10
1000000	0.014	0.004	0.004	0.005	0.009	0.006	0.011	0.004	0.004	0.004
2000000	0.027	0.009	0.015	0.015	0.009	0.008	0.010	0.009	0.008	0.014
4000000	0.045	0.017	0.018	0.020	0.017	0.020	0.017	0.020	0.020	0.017
6000000	0.071	0.025	0.029	0.025	0.025	0.025	0.029	0.030	0.030	0.029
8000000	0.090	0.040	0.033	0.033	0.039	0.039	0.039	0.033	0.033	0.033
10000000	0.116	0.047	0.047	0.047	0.042	0.042	0.045	0.043	0.045	0.048

Table A.4: CPU versus GPU Performance - GPU

Array Size	1	2	3	4	5	6	7	8	9	10
1000000	0.018	0.006	0.006	0.006	0.006	0.007	0.006	0.006	0.006	0.006
2000000	0.060	0.013	0.015	0.017	0.012	0.018	0.012	0.018	0.012	0.018
4000000	0.056	0.029	0.024	0.03	0.030	0.031	0.030	0.030	0.030	0.030
6000000	0.053	0.040	0.040	0.040	0.039	0.040	0.040	0.046	0.041	0.044
8000000	0.077	0.055	0.054	0.056	0.061	0.054	0.052	0.053	0.052	0.053
10000000	0.161	0.156	0.156	0.158	0.155	0.159	0.156	0.156	0.155	0.170

Table A.5: CPU versus GPU Performance - CPU