

Application Acceleration: An Investigation of Automatic Porting Methods for Application Accelerators

A thesis submitted in partial fulfilment
of the requirement for the degree of Doctor of Philosophy

Thomas Henry Outram Beach

**Cardiff University
School of Computer Science &
Informatics**

September 2010



UMI Number: U585458

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



UMI U585458

Published by ProQuest LLC 2013. Copyright in the Dissertation held by the Author.
Microform Edition © ProQuest LLC.

All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code.



ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346


Declaration

This work has not previously been accepted in substance for any degree and is not concurrently submitted in candidature for any degree.

Signed  (candidate) Date 0/5/11


Statement 1

This thesis is being submitted in partial fulfilment of the requirements for the degree of PhD.

Signed  (candidate) Date 0/5/11

Statement 2

This thesis is the result of my own independent work/investigation, except where otherwise stated. Other sources are acknowledged by explicit references.

Signed  (candidate) Date 0/5/11

Statement 3

I hereby give consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed  (candidate) Date 0/5/11

Abstract

Future HPC systems will contain both large collections of multi-core processors and specialist many-core co-processors. These specialised many-core co-processors are typically classified as Application Accelerators. More specifically, Application Accelerations are devices such as GPUs, CELL Processors, FPGAs and custom application specific integrated circuit devices(ASICs). These devices present new challenges to overcome, including their programming difficulties, their diversity and lack of commonality of programming approach between them and the issue of selecting the most appropriate device for an application.

This thesis attempts to tackle these problems by examining the suitability of automatic porting methods.

In the course of this research, relevant software, both academic and commercial, has been analysed to determine how it attempts to solve the problems relating to the use of application acceleration devices. A new approach is then constructed, this approach is an Automatic Self-Modifying Application Porting system that is able to not only port code to an acceleration device, but, using performance data, predict the appropriate device for the code being ported. Additionally, this system is also able to use the performance data that are gathered by the system to modify its own decision making model and improve its future predictions.

Once the system has been developed, a series of applications are trialled and their performance, both in terms of execution time and the accuracy of the systems predictions, are analysed.

This analysis has shown that, although the system is not able to flawlessly predict the correct device for an unseen application, it is able to achieve an accuracy of over 80% and, just as importantly, the code it produces is within $\approx 15\%$ of that produced by an experienced human programmer. This analysis has also shown that while automatically ported code performs favourably in

nearly all cases when compared to a single-core CPU, automatically ported code only out performs a quad-core CPU in three out of seven application case studies. From these results, it is also shown that the system is able to utilise this performance data and build a decision model allowing the users to determine if an automatically ported version of their application will provide performance improvement compared to both CPU types considered.

The availability of such a system may prove valuable in allowing a diverse range of users to utilise the performance supplied by many-core devices within next generation HPC systems.

To my Dad,
Who constantly helped me, encouraged me and even
proof read early chapters of this thesis, but did not live
long enough to see it completed.

Acknowledgements

Firstly, I would like to thank my supervisor; Professor Nick Avis for his help, encouragement, and advice over the course of this project. Secondly I would like to thank Dr Ian Grimstead for all his advice, proof reading, and putting up with my frequent badgering and discussion of ideas (Especially regarding ClearSpeed!)

I would also like to thank other members of the Cardiff Computer Science PGR community, especially those who have been involved in organising the various “FTS” student social events and seminars which have been a welcome break and a pleasant distraction.

Finally, I would like to thank my family and personal friends. My Mum and Dad for supporting me and my elder brother Daniel for providing several valuable pieces of inspiration.

Funded by

EPSRC Engineering and Physical Sciences
Research Council

List of Publications

1. Abstraction of Programming Models Across Multi-Core and GPGPU Architectures - [17]

Euro GPU Mini Symposium as part of the International Conference on Parallel Computing(PARCO) 2009

2. An Intelligent Semi-Automatic Application Porting System for Application Accelerators - [18]

UnConventional High Performance Computing Workshop as part of Conference on Computing Frontiers 2009

3. Poster: An Intelligent Semi-Automatic Application Porting System for Reconfigurable Devices - [16]

Many-core and Reconfigurable Supercomputing Conference (MRSC) 2008

Table of Contents

1	Introduction	1
1.1	Introduction	1
1.2	Hypothesis	5
1.3	Scope	5
1.4	Contributions	6
1.5	Thesis Summary	7
2	Background	10
2.1	Introduction	10
2.2	Graphics Processing Units	11
2.2.1	Device Architecture	11
2.2.2	General Purpose Computation on Graphics Hardware . . .	14
2.2.3	The future of GPGPU	23
2.3	ClearSpeed Acceleration Architecture	26
2.3.1	Device Architecture	27
2.3.2	Programming Tools	28
2.4	Field Programmable Gate Arrays	31
2.5	IBM Cell Broadband Engine	37
2.6	Computational Libraries	39
2.7	Existing Application Acceleration Porting Methods	42
2.7.1	Portland Group Accelerator Compiler	46

2.7.2	Sieve Multicore Programming System	48
2.7.3	HMPP	49
2.7.4	Rapidmind and Ct: C for Throughput Computing	51
2.7.5	Peakstream	53
2.7.6	Evaluation of Existing Methodologies	53
2.8	Chapter Summary	56
3	Overview of the Application Porting System	58
3.1	Introduction	58
3.2	Overall System Architecture	60
3.3	Overview of System Client	64
3.4	Source Parsing and Validation	65
3.5	Kernel Extraction	67
3.6	Kernel Analysis	72
3.7	Validation of Input Kernels	73
3.8	Application Packaging	77
3.9	Chapter Summary	77
4	Code Generation	80
4.1	Introduction	80
4.2	Kernel Selection	81
4.3	Porting to CUDA for the GPU	82
4.3.1	Generation of Host Code	83
4.3.2	Generation of Kernel Code	90
4.3.3	Calculating the execution configuration	93
4.4	Porting to C_N	99
4.4.1	Generation of Host Code	100
4.4.2	Generation of Device Kernel Template	109
4.4.3	Non-Buffered Kernels	110

4.4.4	Buffered Kernels	115
4.4.5	Determining Buffer sizes	119
4.5	Creating Build Scripts	121
4.6	Code Generation Example - GEMM	122
4.7	Chapter Summary	124
5	Device Selection, Self-modification and Expandability	127
5.1	Introduction	127
5.2	Architecture of the Application Classifier	128
5.3	Gathering Metrics	131
5.4	Matching Applications to Devices	132
5.4.1	Making Decisions	134
5.5	Validating Decisions	137
5.6	System Evolution	138
5.6.1	Gathering Performance Data	141
5.6.2	Integration of New Devices	143
5.7	Chapter Summary	144
6	Application Case Studies	147
6.1	Introduction	147
6.2	Seed Applications	153
6.2.1	Structured Grids - Sobel Edge Detector	154
6.2.2	Dense Linear Algebra - Matrix Multiplication	159
6.2.3	N Body Methods	165
6.2.4	Monte Carlo Methods	171
6.3	Test Applications	177
6.3.1	Fast Fourier Transform	177
6.3.2	Canny Edge Detector	188
6.3.3	Iterative Ray Tracer	198

6.4	Classification of Known Applications	209
6.5	Chapter Summary	210
7	Conclusions	215
7.1	Introduction	215
7.2	Research Hypothesis	216
7.3	Contributions	218
7.4	Relation to Current Work	221
7.4.1	HMPP	222
7.4.2	PGI Accelerator Compiler	222
7.4.3	Conclusions	223
7.5	Evaluation of Metrics	224
7.6	ClearSpeed vs NVIDIA GPU	226
7.7	Evaluation	227
7.8	Chapter Summary	229
8	Further Work	232
8.1	Introduction	232
8.2	Making decisions based on other factors	233
8.3	Code Optimisations	234
8.3.1	ClearSpeed:	234
8.3.2	CUDA	235
8.4	Supporting larger data-sets	236
8.5	Supporting Multi-Cards Accelerators	237
8.6	Scheduling	238
8.7	Mapping code to computation libraries	238
8.8	Cloud computing	239
8.9	Adaptive Code Porting	241
8.10	Chapter Summary	242

Appendices	258
A Porting Example: GEMM	258
A.1 Input Source	258
A.2 Code Executing on CPU	261
A.3 CUDA	263
A.3.1 Generated Host Code	263
A.3.2 Generated Device Code	264
A.4 C_N	265
A.4.1 Generated Host Code	265
A.4.2 Generated Device Code: Non Buffered Kernel	270
A.4.3 Generated Device Code: Buffered Kernel	272
B Kernel Description Metrics for FFT Application	278

List of Figures

1.1	Examples of Acceleration Devices: a) CELL B.E [4] b) FPGA from Nallatech [7] c) NVIDIA Tesla [101] d) ClearSpeed [1].	2
1.2	Application Accelerators as Co-processors.	3
2.1	The GPU Rendering Pipeline.	12
2.2	The NVIDIA GeForce 8800 Architecture. Adopted from [45] . . .	13
2.3	NVIDIA CUDA Architecture. Adopted from [46]	20
2.4	The OpenCL Architecture. Adopted from [60]	21
2.5	The Larrabee Architecture, showing 8 CPU like cores, adopted from [122].	25
2.6	The ClearSpeed CSX Architecture. Adopted from [36]	28
2.7	A FPGA Internal Structure adopted from [25]	31
2.8	Typical FPGA design flow adopted from [25]	33
2.9	The CELL/B.E. Architecture adopted from [119].	37
2.10	An Ideal Abstraction for Application Accelerators adopted from [29]	44
2.11	A time line of software released to facilitate development on Application Accelerators	54
3.1	High level design of the Porting system	59
3.2	The Application Porting system	62
3.3	Client in the Compiler Mode	66
3.4	A Control Flow Graph	68

3.5	A Control Flow Graph, labelled with loop IDs	71
3.6	A Control Flow Graph, with Kernels Separated	72
3.7	A Kernel Description File	73
3.8	A Kernel Tree	74
3.9	Client Output	77
3.10	Client in the Execution Step	78
4.1	A Kernel Tree	82
4.2	A Multi-Dimensional Array	86
4.3	A CUBIN file	97
4.4	ClearSpeed: Memory buffer layout	118
4.5	GEMM: Example Kernel Tree	123
5.1	Internal Structure of Application Classifier.	129
5.2	The Database of Performance Data.	130
5.3	An Example Decision Tree.	135
6.1	Graph of Single Precision Execution Times for Structured Grids Application.	155
6.2	Graph of Double Precision Execution Times for Structured Grids Application.	156
6.3	Graph of Single Precision Execution Times for GEMM Application.	160
6.4	Graph of Double Precision Execution Times for GEMM Application.	161
6.5	Graph of Single Precision Execution Times for N-Body Application.	166
6.6	Graph of Double Precision Execution Times for N-Body Application.	167
6.7	Flowchart of the Monte Carlo Technique used based on [115].	172
6.8	Graph of Single Precision Execution Times for Monte Carlo Application.	173
6.9	Graph of Double Precision Execution Times for Monte Carlo Application.	174

6.10	Classification Model Produced after Seed Applications.	178
6.11	Sample input and output of Fast Fourier Transform.	179
6.12	Fast Fourier Transform Application Structure	179
6.13	Classification Model Produced after FFT Application has been profiled.	183
6.14	Graph of Single Precision Execution Times for FFT Application. .	184
6.15	Graph of Double Precision Execution Times for FFT Application.	185
6.16	Sample input and output of a Canny Edge Detector	190
6.17	Canny Application Structure	191
6.18	Classification Model Produced after Canny Application has been profiled.	193
6.19	Graph of Single Precision Execution Times for Canny Application.	195
6.20	Graph of Double Precision Execution Times for Canny Application.	196
6.21	Sample output of Iterative Ray Tracer	198
6.22	Ray Tracer Application Structure	200
6.23	Final Classification Model	203
6.24	Graph of Single Precision Execution Times for Iterative Ray Tracer.	206
6.25	Graph of Double Precision Execution Times for Iterative Ray Tracer.	207
8.1	Cloud Computing.	240
8.2	Adaptive Porting.	241

Listings

2.1	Example of Cg Code adopted from [85]	15
2.2	Example of Brook Code Adopted from [24]	17
2.3	Example of C for CUDA Code Adopted from [46]	19
2.4	OpenCL version of CUDA Code shown in Listing 2.3 based on [70]	22
2.5	Example of C_N Code	30
2.6	Mitrion-C Code - Line numbers added for clarity	35
2.7	PGI Accelerator Compiler Example adopted from [134]	47
2.8	Sieve Example	49
2.9	HMPP Example Codelet Definition	50
2.10	HMPP Codelet Callsite	50
2.11	Rapidmind Example	52
2.12	Ct Code Example	52
3.1	Code Example for Control Flow Graph	68
3.2	Algorithm for Loop Identification	70
3.3	Source code that generated Figure 3.5	71
4.1	CUDA: Initialisation Code	84
4.2	CUDA: Allocating memory	85
4.3	CUDA: Loading a single dimensional array onto the device	85
4.4	CUDA: Loading a multi-dimensional array	87
4.5	CUDA: Calling the device code	87
4.6	CUDA: Loop counter look-up code.	88

4.7	CUDA: Loading data back to host for Single Dimensional Arrays	89
4.8	CUDA: Loading data back to host for a two dimensional array . .	89
4.9	CUDA: Random Number Generation	92
4.10	CUDA: The Kernel Template	93
4.11	ClearSpeed: Initialisation Code	102
4.12	ClearSpeed: Declaring Process Handles	103
4.13	ClearSpeed: Declaring Variable Pointers	103
4.14	ClearSpeed: Loading the Device Code	104
4.15	ClearSpeed: Loading Single Variables	105
4.16	ClearSpeed: Loading Arrays	105
4.17	ClearSpeed: Allocating Memory	108
4.18	ClearSpeed: Calling a Kernel	109
4.19	ClearSpeed: Wait for execution to finish	109
4.20	ClearSpeed: Loading Data Back onto the Host	109
4.21	ClearSpeed: Cleanup	110
4.22	ClearSpeed: Declaring Global Variables	110
4.23	ClearSpeed: Kernel Template	110
4.24	ClearSpeed: Multiple Iterations per PE	111
4.25	ClearSpeed: Random Number Generation	113
4.26	ClearSpeed: Reading from mono memory	114
4.27	ClearSpeed: Writing to mono memory	115
4.28	ClearSpeed: Buffered Kernel Template	117
4.29	CUDA: Makefile	121
4.30	ClearSpeed: Makefile	122
5.1	Algorithm for Integration of New Devices	140
5.2	Timing Code: Starting the Timer	142
5.3	Timing Code: Stopping the Timer	142
5.4	Ensuring CUDA execution finished before stopping the timer. . .	143

6.1 Ray Tracing Algorithm [127] 199

Chapter 1

Introduction

1.1 Introduction

The research described in this thesis centres around the use of Application Acceleration Devices. A device can be classified as an Application Accelerator if it can be added to a computing system to increase the performance of applications running on that system. There are a wide variety of these devices currently available, including:

- Graphics Processing Units (GPU).
- Field Programmable Gate Arrays(FPGA).
- CELL Processors.
- Physics Accelerators.
- Programmable Application Specific Integrated Circuits (ASICS) of which ClearSpeed is an example.
- Audio Processing Accelerators.

With such a wide variety of devices fitting the classification of Application Accelerators, only a subset of these will be considered in this thesis. These are illustrated in Figure 1.1 and will be discussed in more detail in Chapter 2.

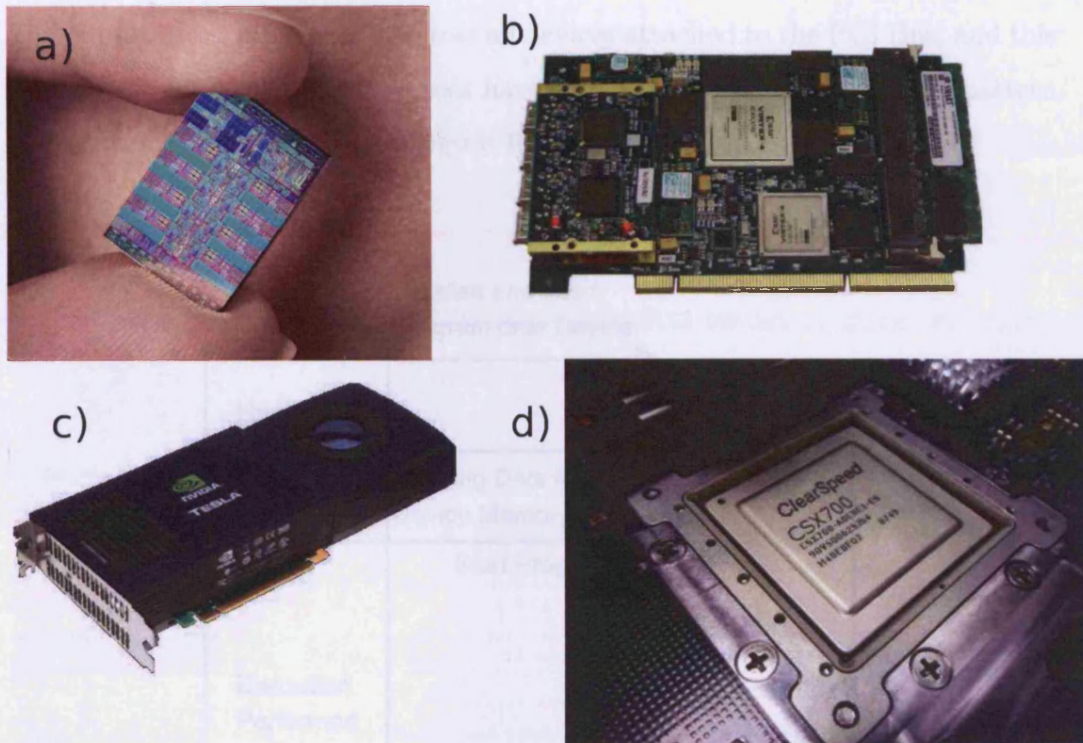


Figure 1.1: Examples of Acceleration Devices: a) CELL B.E [4] b) FPGA from Nallatech [7] c) NVIDIA Tesla [101] d) ClearSpeed [1].

Application Accelerators are generally added to a computing system via the PCI bus and act as co-processors, this has the advantage that tasks executed on the application accelerator are decoupled and can run in parallel with traditional CPU tasks. However, connection via the PCI bus also has several disadvantages:

- Devices are unable to access main memory.
- Devices are unable to access Input or Output devices.

- Devices are unable to access network interfaces.
- Devices are unable to access non-volatile storage devices.

These limitations are common across all devices attached to the PCI Bus, and this means that all Acceleration devices have, at the highest level, a similar pattern of execution. This is shown in Figure 1.2.

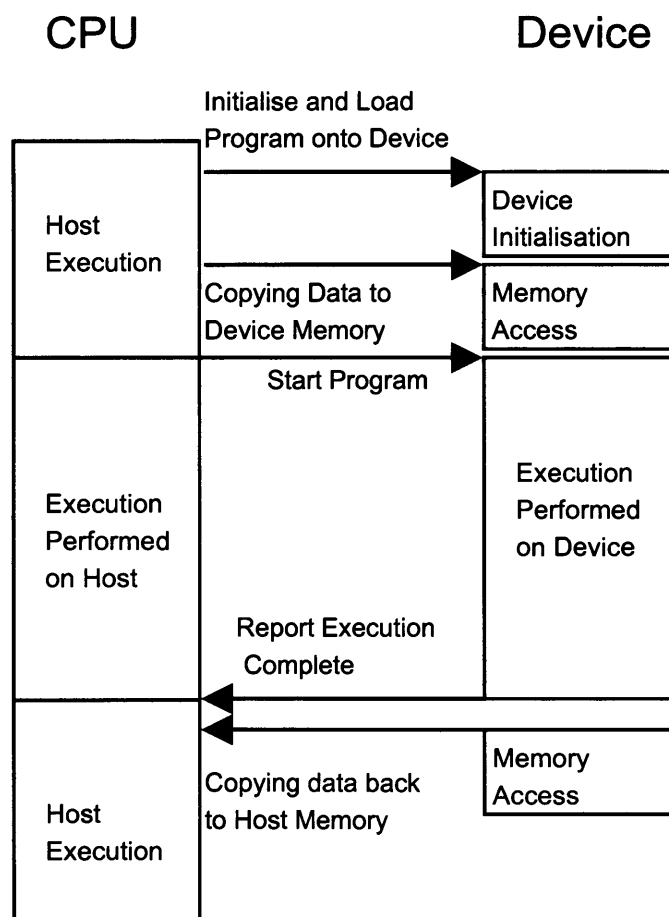


Figure 1.2: Application Accelerators as Co-processors.

Figure 1.2 shows that Application Accelerators have the following overheads compared to standard CPU execution: Device Configuration, transfer of data

from host memory to device memory and transfer of results back to host memory from device memory. All of these overheads add latency and have a detrimental effect on an application's performance. This means that application developers must be mindful of these overheads when developing applications to ensure that the speed-up attained from the use of an application accelerator is sufficient to overcome these overheads.

Additionally, developers using application accelerators must consider that each type of accelerator has a different architecture and different programming style. This can lead to different applications favouring different device types, depending largely on the characteristics of the application and the device.

Finally, a particular device type may have additional advantages/disadvantages that are not related to performance, these could include power usage, compatibility with existing hardware, cost, software development environment and support offered.

These variations between devices and even between manufacturers of the same device, are further discussed in Chapter 2. This wide variations of characteristics does however mean that an application developer must ask themselves the following questions before starting development:

- What device is suitable for accelerating my application?
- What device is suitable for my non-performance based requirements i.e. power requirements?
- Will the performance improvement that results from using this device be worth the time/monetary expenditure?

1.2 Hypothesis

The research hypothesis is:

It is possible to construct a self-modifying and expandable automatic code porting system that can, based on heuristics, select the most appropriate application acceleration device and provide comparable performance to that achievable by an experienced human programmer.

In this hypothesis the term “self-modifying” refers to the ability of the system to improve the heuristic model that it uses to determine the most appropriate device. Additionally, the term “expandable” refers to the ability of the system to be expanded to utilise new acceleration devices, or newer versions of existing acceleration devices.

1.3 Scope

The scope of the research conducted in this thesis can be broken down into three key sections:

Target Users: The level of abstraction provided by the developed system will be targeted specifically at non computer science users. These users will have programming experience but will not be familiar with the intricacies of acceleration devices or their programming methods. The motivation of these users will be to simply achieve application speedup.

Target Applications: The applications considered by the system must already contain at least one section of code that is capable of being executed in parallel.

The research described by this thesis focuses on the locating of parallel code within the input application, deciding on the appropriate method for acceleration of this code and then porting the selected code to the acceleration device. The parallelisation of sequential code is not considered and is deemed beyond the scope of this work.

Target Devices: The target devices that are considered in this thesis are restricted by currently available hardware. The FPGA, CELL, GPU and ClearSpeed accelerators have all been considered, however, due to hardware limitations, only the NVIDIA GPU, and the ClearSpeed accelerator device have been taken forward for development.

1.4 Contributions

This section lists the major contributions of this thesis. The four main contributions are:

1. A novel distributed system and architecture that is able to analyse and port input applications to an acceleration device and, with a reasonable success rate, predict the most appropriate device for the application concerned.
2. Demonstration that the system is able to modify itself, in that it is able, from experience, to adapt the model that is used to select an acceleration device and that it is able to adapt to the introduction of the new devices, or improved versions of existing devices.
3. The ability to demonstrate, through the use of well understood and developed machine learning techniques, a set of explicit parameters and features that can be used to describe the selection of an appropriate acceleration device.

4. The provision of a route to application acceleration to non-computer science users. This may be the porting of an application, generating an efficient initial port from which further performance improvements can be made by experienced human programmers, or determining in a quick and simple manner, whether the users application is suitable for acceleration.

1.5 Thesis Summary

Chapter 2: Background

Chapter 2 introduces the field of application accelerators, outlining the different devices that are currently available and showing the differences between them. This chapter also examines the programming methods that these devices support and shows that the differences between these programming methods can vary from as little as a different API to having to use a different programming paradigm. Using this knowledge, this chapter then describes the problems that are associated with programming acceleration devices, namely: the difficulty in programming such devices and the lack of abstraction between devices. Finally, this chapter will describe and compare current academic and commercial systems that are also aiming to tackle these problems and discuss their successes and failures.

Chapter 3: Overview of the Application Porting System

Chapter 3 outlines the overall structure of the system that has been constructed to validate the hypothesis outlined in Section 1.2. This chapter will describe the distributed nature of the system and how the components of the system fit together. Additionally, this Chapter will describe the client that has been developed to allow users to interface with the porting system.

Chapter 4: Code Generation

Chapter 4 describes the code generation functionality of the system. This chapter will describe the process that is used to port code to C_N for ClearSpeed and CUDA-C for NVIDIA GPUs. This chapter will also illustrate the differences between these two programming methods.

Chapter 5: Device Selection, Self-Modification and Expandability

Chapter 5 will describe the remaining functionality of the application porting system. This chapter will explain how the system is able to perform the decision making required to match an application to the most appropriate acceleration device. Secondly, this chapter will describe how the system is able to achieve its goals of being “self-modifying” and “expandable”.

Chapter 6: Application Case Studies

Chapter 6 describes a series of applications that have been selected to test the system. These applications are divided into two categories. A series of relatively simple “seed” applications are trialled in order to provide a base of knowledge for the system to operate with more complex examples.

Three more complicated applications are also executed. For each of these applications the performance and the decisions taken are analysed in order to judge the effectiveness of the system. Additionally, by the execution of these applications in sequence, the ability of the system to modify its classification model will be tested. Finally, the performance of the generated code will be tested against the performance of an optimised, but not re-factored, hand port produced by the developer.

Chapter 7: Conclusions

Chapter 7 will outline the overall conclusions that can be drawn from this work. This chapter will analyse the results from Chapter 6 against the hypothesis presented in Chapter 1 and then validate the contributions that were also presented in Chapter 1.

Secondly, this chapter will evaluate the work done in this thesis, examining the relative strengths and weaknesses of the work in isolation and in comparison to the two most relevant commercial products.

Finally, this chapter will also present conclusions on the relative differences between the two acceleration devices that have been utilised in the course of this research.

Chapter 8: Future Work

Chapter 8 will describe ideas for the future development of this work. This will largely centre around suggestions for the future development of the application porting system that has been developed. In addition to suggested modifications it will present early thoughts on how the system can be deployed in a cloud computing environment.

Background

2.1 Introduction

This chapter aims to introduce the key elements of this research, focusing specifically on prior work and approaches related to the work that is presented in this thesis. This chapter will outline the types of acceleration devices that are currently available, expanding on the brief introduction presented in Chapter 1. In particular this chapter will cover the development of the hardware architecture of these devices. It will also provide an overview of the software tools, both academic and commercial, that have been developed to facilitate their use in the field of High Performance Computing. This chapter will then discuss some of the successes that have been achieved in porting algorithms to these devices and developing numerical libraries to support the work of other developers. Finally, this chapter will state the key issues currently preventing the adoption of these devices and examine the currently available industrial and academic solutions. This examination of existing technology will enable a comparison of the available solutions and the identification of key areas for improvement.

2.2 Graphics Processing Units

While probably the most recent device to find acceptance as a viable Application Accelerator for High Performance Computing (HPC) applications, the Graphics Processing Unit (GPU) has experienced an explosion of interest, driven by the fact that currently GPUs represent some of the most computationally powerful hardware for the dollar [111]. This has caused a vibrant community of developers to emerge. For example, considering only the use of NVIDIA's tools, there have already been over 2700 citations on Google Scholar and 300 universities worldwide are now teaching GPU programming using NVIDIA CUDA [50].

The GPU market is dominated currently by two companies: NVIDIA and AMD (formerly ATI), although Intel were, for a time, attempting to compete with their Larrabee [122] architecture, although this product, as a GPU, has now been cancelled. This section will examine the development of the GPU architectures and programming models employed by both NVIDIA and AMD. It will also discuss briefly existing work that has been conducted in the use of the GPU in accelerating HPC applications.

2.2.1 Device Architecture

The Graphics Processing Unit (GPU) was originally designed as a fixed function rendering pipeline (shown in Figure 2.1). Within this pipeline the Vertex and Fragment processing units have been historically configurable but not programmable [110]. As GPUs have evolved each new generation has added to the functionality of these Vertex and Fragment processing stages. The first programmable stage was introduced in 1999 when NVIDIA added support for register combiner operations which allowed a limited programmability [111]. A

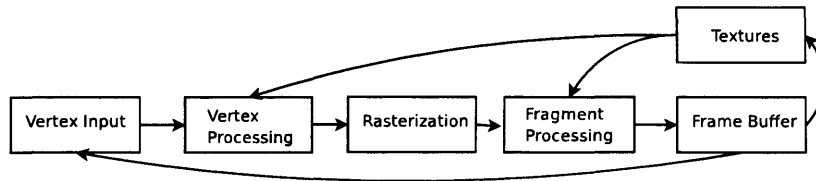


Figure 2.1: The GPU Rendering Pipeline.

further key development was the introduction of an assembly language that allowed the specification of programs that run on a per vertex basis [84]. With the addition of support for fragment (pixel) shading as well as vertex shaders, this programmability was formally defined as a Shader Model. Shader Models have improved through several iterations, with more complete instruction sets, more flexible control-flow operations and larger limits on the size and resource consumption of their programs [110]. This process of improvement, without radically altering the architecture, continued until the development of Shader Model 3.0 in 2004.

Shader Model 4.0, first made public in 2006, was the first shader model to present a unified architecture rather than utilising separate vertex and fragment processing units. This new architecture allowed the unification of the instruction sets by defining a single common core, with a virtual machine, as the base for each of the programmable stages. This new unified model is considerably closer to providing all the arithmetic, logic and flow control constructs available on a standard CPU [20].

The first GPU to utilise the unified shader architecture defined by Shader Model 4.0 was the Xenos chip by AMD in 2005. This chip was used solely in the XBox 360 [110]. However NVIDIA were the first to make Shader Model 4.0 cards widely available with their GeForce 8800 in 2006. The GTX version of this card comprised 128 of these unified processors (dubbed stream processors by NVIDIA) [45]. All the programmable and fixed function aspects of the graphics pipeline

are now computed on these stream processors. This allows more complex load balancing to take place, allowing any stage of the pipeline to consume more of the available stream processors dependent on its requirements. This load balancing is critical to prevent bottlenecks caused by stages of the graphics pipeline taking longer to compute than others.

Currently all new GPUs produced by the two main vendors, NVIDIA and AMD, utilise this unified architecture, of which NVIDIA's version is shown in Figure 2.2.

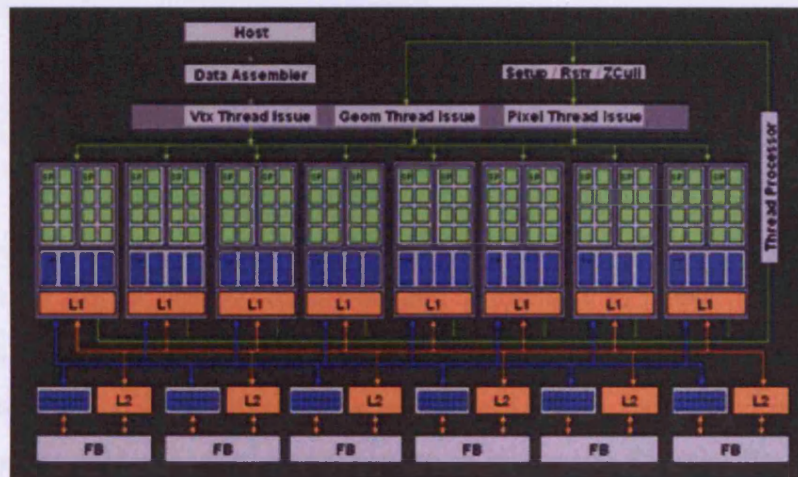


Figure 2.2: The NVIDIA GeForce 8800 Architecture. Adopted from [45]

One of the most recent developments in graphics hardware was the addition of support for double precision floating point arithmetic. The FireStream 9170, launched in 2007, was the first card on the market to support double precision. NVIDIA soon followed with their double precision chipset, the GTX 200 [47] and double precision support has now become the standard for high end, professional level GPUs. However, it should be noted that utilising double precision comes with a performance penalty as all current model GPUs have fewer double precision processing units when compared to single precision processing units. e.g. the

NVIDIA Tesla C1070's double precision performance is approximately 10% of its single precision performance [44].

GPUs have advanced considerably, even in the last few years. Current GPU architectures bear little resemblance to those of ten years ago and GPUs are now no longer a fixed function implementations of the graphics pipeline but are now fully programmable devices with supporting fixed functionality enabling it to carry out its primary role as a rendering engine.

2.2.2 General Purpose Computation on Graphics Hardware

One of the first uses of the Graphics Processing Unit as a General Purpose computation device can be found in [26], published in 1994. In this paper, Cabral et al implemented Filtered Backprojection, a tomographic reconstruction algorithm, on the GPU by utilising available texture mapping hardware. In order to achieve this, Cabral was forced to re-factor the Filtered Backprojection algorithm into a graphics problem, within the constraints of the fixed-function OpenGL pipeline available at the time. The result of this was a two pass rendering algorithm that utilising the GPU's hardware accelerated texture mapping and frame buffer accumulation functions. This new approach gave performance in excess of 100x faster than executing on the CPU.

Despite these early uses of the GPU for general purpose computing, general purpose graphics processing units(GPGPU) were still largely impractical for developers. At the time, there were simply no available tools to allow developers to leverage the power of a GPU unless their application was able to be refactored to leverage on the fixed-function OpenGL pipeline. Interest was renewed with the

development of programmable shaders and the formalisation of these as Shader Models. Several high level programming languages for shaders originated from this formalisation process. HLSL from Microsoft [113], GLSL for OpenGL [76] and Cg from Nvidia [85].

Listing 2.1: Example of Cg Code adopted from [85]

```
float4 main(appin IN, out float4 COUT, uniform Light
lights []) {
    for (int i=0; i < lights.Length; i++) {
        Cl=lights[i].illuminate(IN.pos,L);
        color+=Cl * Plastic(texcolor,L,Nn,In,30);
    }
    COUT=color;
}
```

An example of Cg code is shown in listing 2.1 and the resemblance to C can be easily seen. Both HLSL and GLSL are locked to their respective graphics implementations while Cg provides the ability to compile to other targets. However, all these languages share one main disadvantage: they are at their core, shading languages. So all general purpose programming must be mapped onto graphical concepts [64] i.e.:

- Textures → Arrays.
- Render to Texture → Feedback.
- Fragment Programs → Inner loops.
- Geometry Rasterization → Computation Invocation.
- Texture Coordinates → Computation Domain.

– Vertex Coordinates → Computation Range.

This means that programming using these languages is vastly different from standard programming methodologies and requires programmers, unless already familiar with graphical programming, to undertake various unfamiliar tasks such as drawing geometry and manipulation of the camera to achieve the desired computation steps.

Despite these problems, the promise of increased performance made by the GPU encouraged a considerable amount of work in the field and a large community grew up around the “GPGPU” (General Purpose Graphics Processing Units) phenomena.

Many items of literature have dealt with the porting of applications to the GPGPU using Shader Model 3.0 methods. In 2003 Moreland et al, ported the Fast Fourier Transform to the GPU [94] and their implementation performed competitively with highly optimised CPU implementations. Other applications that have been successfully implemented on the GPU include Ray Tracing [118], Volume Rendering [124] [126] and a re-implementation of earlier work on Filtered Backprojection using commodity hardware and more modern Shader Model 3.0 techniques [135].

It was commonly accepted that shader programming was a woefully inadequate solution for General Purpose Computation on GPUs and several pieces of work were conducted in an attempt to resolve this. Lefohn has designed Glift which defines a set of high level GPU data structures [82]. Glift builds on Cg from NVIDIA to provide a set of random access data structures, similar to those provided the STL(Standard Template Library) for C++. The Brook streaming language was developed as a further attempt to solve these problems. The Brook language presents a streaming model to the programmer with the Brook compiler

and runtime system provides the mapping onto the previous discussed GPU APIs [24]. The Brook language enables the programmer to represent their program in terms of streams and kernels. The language also supports many additional features often desired by newcomers to GPGPU programming such as reductions. An example of the Brook programming language is shown in listing 2.2

Listing 2.2: Example of Brook Code Adopted from [24]

```
kernel void mul( float a<>, float b<>, out float c<>) {  
    c= a * b;  
}  
  
//call the kernel  
  
float a<50>;  
float b<50>;  
float c<50>;  
  
mul(a, b, c);
```

Brook was generally successful and several applications were successfully ported using it. In 2006 Elsen et al, ported a N-Body simulation to the GPU using Brook and they achieved, in some cases, upwards of 25x speedup [53].

Several other abstractions have been developed to fill a similar space of providing translation from high level languages to shader based languages. SH [5] was developed by a team at the University of Waterloo and was eventually commercialised and became Rapidmind, which is discussed in Section 2.7.4. PyGPU was developed to add GPU acceleration to Python [83] and Microsoft developed their Accelerator language which is a set of additions to C# utilising a special data type called Parallel Arrays [129]. While these higher level languages go some way to solving the problems of programming in a shader based environment they cannot escape the fact that the architecture they are

abstracting from is still too specific to GPU concepts to allow true general purpose programming.

However, there were two developments in GPGPU that changed all of this. Firstly, the introduction of Shader Model 4.0, which allowed programming of the GPU in a non shader based way allowed the introduction of new high level tools, such as CUDA and CTM, which are discussed below, and the further improvement of existing tools.

Secondly, one of the first languages to fully utilise this new unified model was introduced: NVIDIA's C for Compute Unified Device Architecture language (C for CUDA). This language leverages on the new functionality provided in Shader Model 4.0 to provide what is probably the most popular language for General Purpose development on GPUs.

C for CUDA allows programming in a full implementation of the C language, with some GPU extensions and an host processor API. C for CUDA provides the programmer with the following features [46]:

- Methods for on card memory management.
- Ability to define methods for execution on CPU or device.
- Kernel Invocation methods.
- A method to specify the allocation of threads across the device.

The CUDA architecture allows the user to declare a kernel which is executed N times by N threads. These threads are grouped together into thread blocks. All threads within a thread block can cooperate together, sharing memory and synchronising. These thread blocks are then arranged into a two dimensional

grid of thread blocks. Each thread block must be able to execute independently of any other. The number of thread blocks directly effects the parallelisation of the application across the multiple processing units available on the GPU. This architecture is shown in Figure 2.3 and an example of C for CUDA is shown in listing 2.3.

Listing 2.3: Example of C for CUDA Code Adopted from [46]

```
--global__ void mul(float * a, float *b, float *c) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N) c[i][j] = a[i][j] + b[i][j];
}

int main() {

    cudaSetDevice(0); // use first GPU
    cudaMalloc (.....); // allocate memory
    cudaMemcpy (.....); //copy to device

    mul<< dimensions of grid , dimensions of each
        thread block>>(A,B,C);

    cudaMemcpy (.....); // copy results from device
    cudaFree (.....); //free memory from device.

}
```

AMD have also constructed a software stack to utilise the new Shader Model 4.0 architecture. Their initial offering was the Close to the Metal(CTM) assembly language. CTM was designed and marketed as a low level language, with a supporting API; this discouraged its adoption by many programmers who were seeking to develop at a higher level of abstraction. The CTM system allowed direct access by the programmer to the floating point processors inside the card [72] via an assembly language interface. AMD then expanded their offering

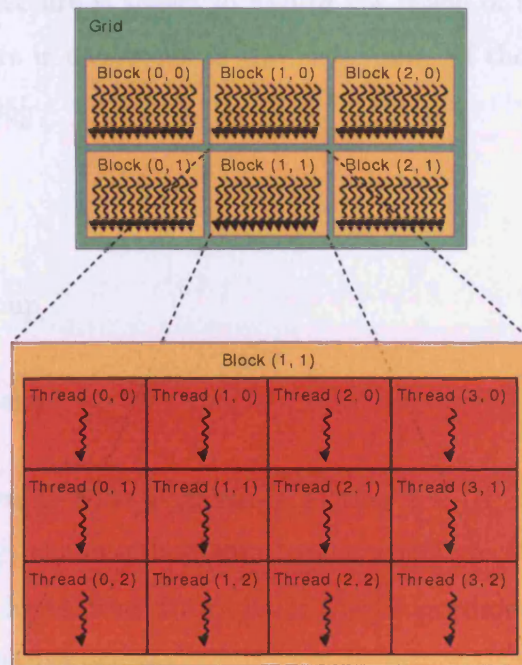


Figure 2.3: NVIDIA CUDA Architecture. Adopted from [46]

by producing their Stream Computing Software Stack. This stack consisted of the AMD Compute Abstraction Layer, a development of CTM and a high level language based on Brook called Brook+ [71].

In addition to NVIDIA CUDA and Brook+ from AMD, one of the more recent developments has been the introduction of OpenCL. OpenCL (Open Computing Language) is an open, royalty free standard for general purpose programming across CPUs, GPUs and other processors. It has been developed by the Khronos Group [61].

The OpenCL standard is based on C99 and aims to allow programming of computation devices while abstracting from the underlying hardware. In general design terms OpenCL is similar to CUDA, although its API is of a slightly lower level to allow for the differing characteristics of the devices on which it targets.

The OpenCL architecture is shown in Figure 2.4. Each of the basic units of the OpenCL architecture is analogous to the basic units of the CUDA architecture discussed earlier [105]:

– Grid - NDRange.

– Block - Work-Group.

– Thread - Work-Item.

An example of OpenCL code is shown in Listing 2.4. It can be seen from this listing, that although the overall architecture is similar to NVIDIA's CUDA, the API operates at a lower level thus giving the programmer more flexibility to program with alternative devices.

Currently OpenCL is supported by AMD, NVIDIA and Apple. At the time of writing, however, all current OpenCL implementation are for GPUs only.

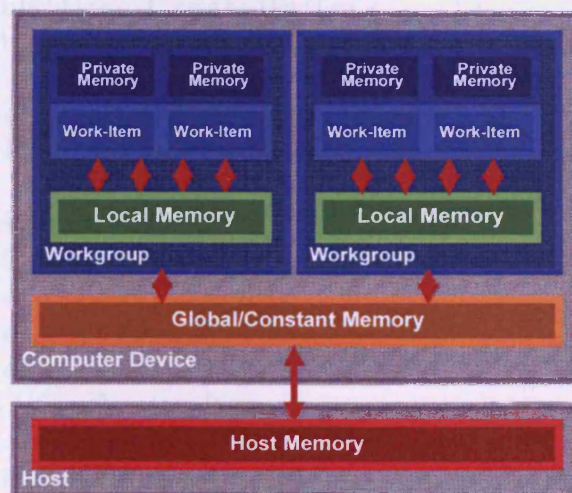


Figure 2.4: The OpenCL Architecture. Adopted from [60]

Listing 2.4: OpenCL version of CUDA Code shown in Listing 2.3 based on [70]

```
//using runtime compilation so source stored in character  
array  
char source[]=  
kernel void mul(global float * a,global float *b,global  
float *c) {  
int id=get_global_id(0);  
c[id] = a[id] + b[id];  
}  
;  
int main() {  
  
    //setting up the device  
    clGetDeviceIDs(NULL,CLDEVICE_TYPE_GPU,1,&device ,  
        num_devices_returned);  
    context=clCreateContext(0,1,device ,NULL,NULL,&err  
        );  
    queue_gpu=clCreateCommandQueue(context , device ,0,&  
        err):  
  
    //create memory on device  
    memobjs[0]=clCreateBuffer(....);  
    memobjs[1]=clCreateBuffer(....);  
    memobjs[2]=clCreateBuffer(....);  
  
    //load the program onto device  
    program=clCreateProgramWithSource(context,1,&source  
        ...);  
    clBuildProgram(program,.....);  
    kernel=clCreateKernel(program,"mul",NULL);  
    clSetKernelArg(kernel,0,....);  
    clSetKernelArg(kernel,1,....);  
    clSetKernelArg(kernel,2,....);  
    global_work_size[0]=n;  
    //run program  
    clEnqueueNDRangeKernel(cmd_queue, kernel,.....);  
  
    //read data back  
    clEnqueueReadBuffer(context,memobjs[2],.....);  
}
```

The competing implementations; CUDA, OpenCL and Brook+ have all enjoyed success and have succeeded in making GPGPU programming available to a far wider community than before. However, there are still problems. It is commonly agreed that in order to make GPU computing more accessible higher level APIs are needed [31] and several projects have been undertaking looking into this. Breitbart [23] has developed a C++ framework designed to allow the easy integration of CUDA into existing C++ applications. Additionally Ueng et al [131] have developed CUDA-Lite. CUDA-Lite is a tool that uses a series of programmer supplied annotations within a CUDA C code program that uses only the device's global memory. CUDA-Lite performs a set of automated transformations to this input to produce output code that maximizes memory performance via memory coalescing and ensuring that maximum use is made of all levels of CUDA's memory hierarchy. The different levels of this hierarchy, such as on-chip shared memory, are often ignored by developers due to a lack of understanding of the CUDA memory hierarchy, which is substantially different from a standard CPU, and the possible performance improvements that can be gained from its correct use. This problem of abstraction is currently being addressed in a variety of ways by tools such as HMPP and Rapidmind, which are discussed in detail in section 2.7.

2.2.3 The future of GPGPU

As one of the most rapidly advancing application acceleration devices, new and improved GPUs are regularly being developed by the major device vendors to enable them to keep their competitive edge, both in the GPGPU market and in the traditional graphics card market.

The latest development from NVIDIA has been the FERMI architecture. FERMI

is touted as the Next Generation CUDA Architecture and NVIDIA boasts a 4.2x performance improvement over the previous CUDA chipsets [48]. NVIDIA FERMI also promises the following key areas of improvement:

- Improved Double Precision Performance,
- ECC Support,
- Faster Context Switching,
- 4x More CUDA Cores per Multiprocessor,
- An Unified Address Space allowing the support of C++, including pointers,
- Allows the scheduling of up to four concurrent kernels.

If FERMI delivers on these promises, then it is anticipated that it will bring large performance improvements to GPGPU users, however, it is not the architectural leap that was encountered in the transition to Shader Model 4.0.

The other highly anticipated development in the field of GPGPU, was the release of the now discontinued Larrabee processor from Intel [122]. The philosophy behind Larrabee was that of a graphics card built up of CPU like cores running an extended version of the x86 instruction set. Intel believed that this CPU-like, x86 based architecture would allow for more flexibility than offered by current GPUs [122]. A diagram of the Larrabee architecture is shown in Figure 2.5.

The Larrabee architecture itself consisted of a set of multiple instantiations of an in-order CPU core that is augmented by a wide vector processor. These cores all communicate via a high bandwidth interconnect network. The Larrabee architecture only contains the minimum amount of fixed function logic to enable

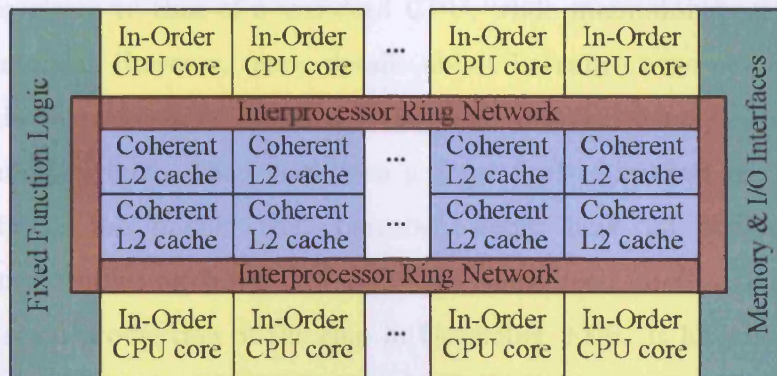


Figure 2.5: The Larrabee Architecture, showing 8 CPU like cores, adopted from [122].

it to perform its graphics based tasks. The only fixed function logic that was expected to be included was texture filtering.

The HPC Development model for Larrabee, known as Larrabee Native, consisted of a complete C/C++ compiler that allowed static compilation of many existing C/C++ applications to Larrabee. However, there are several additional libraries that had to be used to gain the maximum benefit from Larrabee:

- Libraries to provide communication between host and device card. These allowed fast message passing between Larrabee and the host CPU. In addition both synchronous and asynchronous data transfers were to be supported.
- Threading using Larrabee's native software threading capability, exposed via a POSIX Threads API.
- SIMD vectorization. Each of Larrabee's SIMD processors were fully programmable by the programmer, in addition the native Larrabee compiler included a version of Intel's auto-vectorization compiler.

Larrabee promised a major architectural shift for GPUs, bringing their archi-

itecture far closer to that of a standard CPU, while maintaining and improving on performance. However, many details about Larrabee were never confirmed, such as the number of cores on the card and detailed examples of the HPC programming system. There had been a great deal of interest in the scientific community in the impact that Larrabee would have on accelerating high performance computing, however, despite the widespread interest, Intel cancelled the large scale production of the chip in December 2009. It has, however, been announced that Intel have no plans to release Larrabee as a discrete graphics chip, but they will be investigating its use as a HPC product [77] under the name “Knights Ferry” and the forthcoming “Knights Corner” products [52].

2.3 ClearSpeed Acceleration Architecture

ClearSpeed Inc. was founded in 2001 and originated from PixelFusion, a company that developed parallel technologies for high end graphics. At the core of ClearSpeed’s offering was their acceleration chip, which evolved over several iterations. The CS301 was ClearSpeed’s first chip and was released to customers in 2004. This was followed by the CSX600 in 2005 and the CSX700 in 2008 [87]. Through all these iterations ClearSpeed kept a similar architectural model in their chipsets. ClearSpeed accelerators have traditionally had several key advantages over other acceleration devices such as GPUs:

- IEEE 754 floating point compatibility.
- Low power usage $\approx 10\text{W}$.
- ECC Memory.

These advantages allowed ClearSpeed to achieve some success in the HPC market, with several applications being successfully ported to the device, such as: BUDE, MolPro, Amber 9, Monte Carlo Codes, and CFD(Computational Fluid Dynamics) methods [33] [34] [87].

However, in recent years the GPU manufacturers have released cards that have slowly eroded ClearSpeed's key advantages while offering better performance. ClearSpeed also suffered due to the onset of the recession in 2009, this affected ClearSpeed's largest market, the financial sector. These two factors have led to a sharp decline in business for ClearSpeed in the period up to the end of 2009 and they ceased trading in 2010.

The ClearSpeed architecture along with the Software Development Environment that ClearSpeed provides are now described in more detail in the following sections.

2.3.1 Device Architecture

The CSX Chip architecture is illustrated in Figure 2.6. The architecture itself follows a SIMD(Single Instruction Multiple Data) pattern. It consists of a RISC-like control unit and a parallel execution unit. The parallel execution unit consists of a set of 96 Poly Processing Elements. These poly units operate in synchronous mode and are responsible for the execution of instruction in a SIMD manner [36]. Each Poly Processing Element, which provide the parallel processing power of the device, contains several functional units such as: Floating point units, Integer multiply-accumulate units and arithmetic units. Each PE(Processing Element) also contains its own registers, a small amount (6kB) of fast private memory, a data path to the device's main memory and the ability to communicate with neighbouring PE's using an operation known as Swazzle.

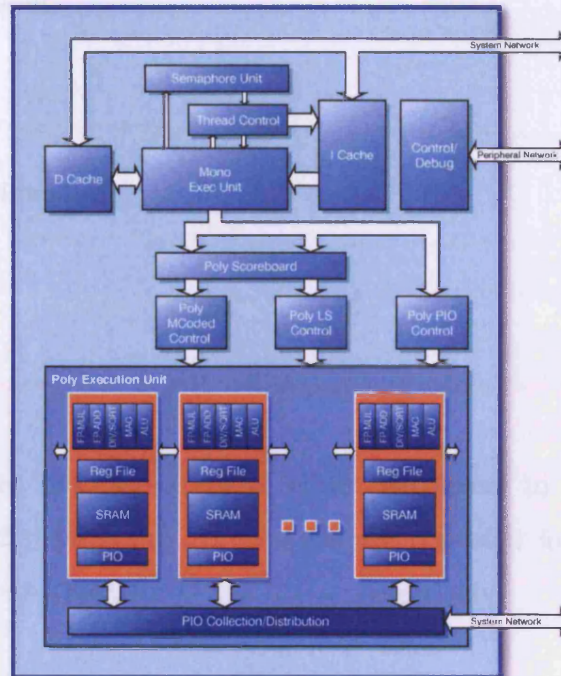


Figure 2.6: The ClearSpeed CSX Architecture. Adopted from [36]

Aggregated across all the parallel processing units, the last generation of the CSX chip boasted the following performance statistics [36]:

- Over 25 GFlops processing power, in single or double precision.
- 25,000 MIPS (Millions of Instructions Per Second).
- 96 Gbytes/s internal memory bandwidth.

2.3.2 Programming Tools

ClearSpeed provided a very complete development kit for their architecture, consisting of the following components:

- Compiler tool-chain,
- Libraries,
- Instruction set simulator,
- Debugger,
- A disassembler.

Their programming interface consists of an extension to C, called C_N for programming on-chip programs. Two APIs are provided for programming on the host; CSAPI and CSPX for C and C++ respectively.

CSAPI provides standard functionality for loading programs, copying data to/from the device and querying device characteristics, while CSPX provides several items of additional functionality and a higher level approach, allowing programs on the device to be called using a RPC(Remote Procedure Call) mechanism [75].

The main addition in the C_N language is the ability to distinguish between variables stored in the device's main menu (mono variables) and in the memory of a specific poly processor (poly variables) using two keywords: *mono* and *poly* [74]. An example of a C_N program is shown in Listing 2.5.

This program illustrates the main features of the C_N language. It shows how data are loaded from the array *data*, which is stored in mono memory, into the variable *a* which is stored in poly memory. The datum is loaded from the correct location in the array by using the *get_penum* method. This method returns a value between 0 and 95 depending on which processing element the code is executing

Listing 2.5: Example of C_N Code

```
mono float *data;
int main(int argc, char**argv) {
    short SEMAPHORE=1;
    poly float a;
    async_memcpy2p(SEMAPHORE,&a, data+get_penum(),
        sizeof(float));
    sem_wait(SEMAPHORE);
    a+=20;
    async_memcpy2m(SEMAPHORE, data+get_penum(),&a,
        sizeof(float));
    sem_wait(SEMAPHORE);
}
```

on. This example also shows how semaphores can be used to cause the program to block while waiting for memory transfers to complete [35].

Additionally the ClearSpeed SDK also contained a set of tools to support programming. A full debugger was available, which included the ability to step-through and monitor the program internals on the device itself. The SDK also contained a cycle accurate simulator, which allowed the testing and debugging of C_N programs without having a Clearseed device present.

ClearSpeed, as a fully commercial product, was a uniformly developed application acceleration platform. There is a single card programming API available, with two host APIs. While there have been many applications successfully ported to the device, the commercial nature of the device, and its pricing, has led to it being less widely adopted than accelerators such as the GPU.

2.4 Field Programmable Gate Arrays

Field Programmable Gate Arrays or FPGAs were invented by Ross Freeman in the mid-1980s [25]. An FPGA is a semiconductor device that consists of programmable logic elements, interconnects and IO(Input/Output) blocks. All of these elements are configurable at run time. This provides the FPGA's key advantage over ASIC(Application Specific Integrated Circuit) devices: allowing complex digital circuits to be constructed on the FPGA at run-time. The internal structure of an FPGA is shown in Figure 2.7.

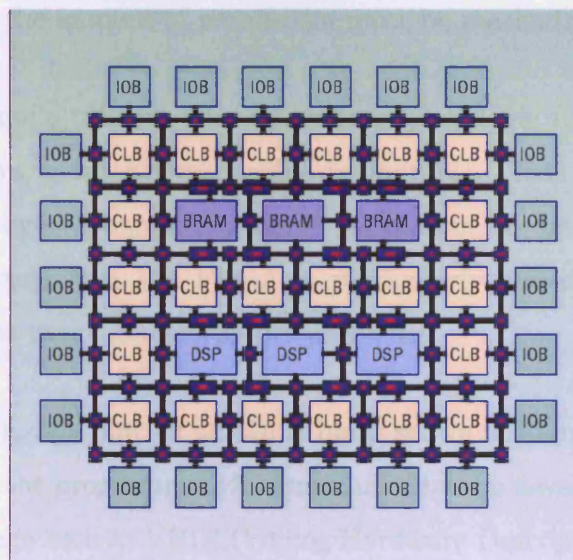


Figure 2.7: A FPGA Internal Structure adopted from [25]

Initially the FPGA was mainly used for prototyping purposes, however in recent years FPGAs have begun to contain enough programmable logic resources to enable them to be of interest to the HPC community. Many systems companies produce FPGA based supercomputers, including Cray, SRC, and more recently Nallatech [132].

However, as with many Application Acceleration devices, the biggest barrier to adoption is the programming method. FPGAs have arguably the most radically different programming method out of all acceleration devices and there are many additional considerations when developing for the FPGA:

- The internal clock speed of an FPGA is many times slower than that of a standard CPU. The performance has to come from performing many operations in parallel per clock cycle.
- There is only limited logic units available on an FPGA so a balance between program size and the amount of parallelism must be reached.
- Floating point units take up a far larger amount of logic space on the FPGA, so avoiding floating point where possible is desirable. This enables either the execution of a larger program, or more parallelism to be achieved. These characteristics means that the FPGA is particularly suited to integer based algorithms, such as many Bioinformatics problems.

An overall FPGA design flow for an algorithm is shown in Figure 2.8. This design flow assumes that the programming language used by the developer is a hardware description language such as VHDL(Verilog Hardware Description Language). It is important to note that even once the application is in a hardware description language, there is still a large amount of testing and validation to be done before it can run on the device. The key step in this process is the place and route algorithm, this algorithm can take many hours to execute and will not always converge into a valid design, meaning that the process must be repeated.

Programming in VHDL is programming at an individual gate level, which is completely impractical for the majority of the HPC community. This had led the development of many higher level languages and compilers to facilitate general

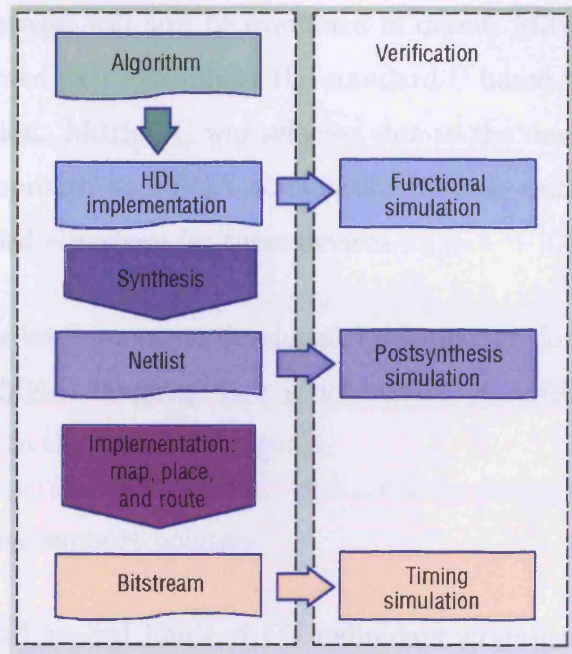


Figure 2.8: Typical FPGA design flow adopted from [25]

purpose computing using these devices, these include:

- Mitrion-C [92].
- DIME-C [55].
- Handel-C [132].
- Trident C to VHDL Compiler [130].
- Single Assignment C (SA-C) (A modification of C where each variable may only be assigned once) [95].
- ASC: A Stream Compiler. (C++ with the addition of several user-defined types, macros, operators and function calls) [89].

Two of these languages will now be examined in detail: Mitrion-C and Dime-C. DIME-C was selected as it exemplifies the standard C based high level languages used for compilation. Mitrion-C was selected due to the developers, Mitronics, taking a novel approach to FPGA compilation that is unlike the majority of languages developed elsewhere for these devices.

DIME-C is a high level language developed by Nallatech for their HPC FPGA devices. The DIME-C language is a strict subset of ANSI C. The following restrictions apply to the DIME-C language:

- DIME-C does not support pointers.
- DIME-C has had several items of C's redundant grammar removed, namely certain switch statements and loops.

DIME-C also had several additions to facilitate compatibility with the FPGA in the form of function calls that allow the following functionality:

- A memory access library that allows efficient access to external SDRAM memories on the FPGA board.
- A math library.
- Functions to provide access to FIFO channels.

In conclusion, DIME-C provides the programmer the ability to develop for the FPGA in a familiar syntax. However, there are still significant differences and limitations that make development of efficient programs in DIME-C a difficult task and, as with all Acceleration devices, an understanding of the underlying hardware is required to selecting an appropriate application for porting and to make maximum use of the device.

Mitrion-C takes a radically different approach to any existing programming method. The Mitrion-C language, while very similar to C syntactically, is closer to a functional language in its method of execution. The Mitrion-C language is a single-assignment language centred around data-dependencies. So the order in which statements appear in the program, is largely irrelevant. A statement is executed as soon as all its data dependencies are available. In the trivial examples shown in Listing 2.6; lines 4 and 6 will be executed in parallel while line 5 will need to wait for line 4 to execute before it can be executed.

Listing 2.6: Mitrion-C Code - Line numbers added for clarity

```
1: int:8 a;  
2: int:8 b;  
3: int:8 c;  
4: c=5*4;  
5: a=c*2;  
6: b=5*3;
```

As parallelism is implicit in the Mitrion-C language, it provides two different types of loops; *for* will execute the body of the loop in an iterative manner and *foreach* will execute a loop in a parallel manner.

Mitrion-C executes its code via a virtual machine. The virtual machine is configured using characteristics extracted from the Mitrion-C source code. The virtual machine code, together with the Mitrion-C program code are then compiled into VHDL and then placed onto the FPGA [91].

In addition to programming languages mentioned above, several other pieces of software have been developed to assist development with FPGAs. Holland et al. [66] have developed a Reconfigurable Amenability Test(RAT) to enable the estimation of the applications compatibility for porting to a FPGA and

Koehler et al [78] have proposed a performance analysis framework, which by using instrumentation of the device code, allows the monitoring of the device's performance at runtime. Andrews et al [13] have developed a system known as HThreads which is a pthreads compatible library that allows specified threads from the pthreads application to be compiled either for the CPU or FPGA. Another library that has been developed to aid FPGA programming is the Vforce library [93]. Vforce is an extension of the Vector Signal and Image Processing Library, that provides a FPGA hardware implementations for the algorithms in the library, encapsulating them beneath the libraries standard API. This allows users of the library to develop for the FPGA without the use of any hardware specific implementation.

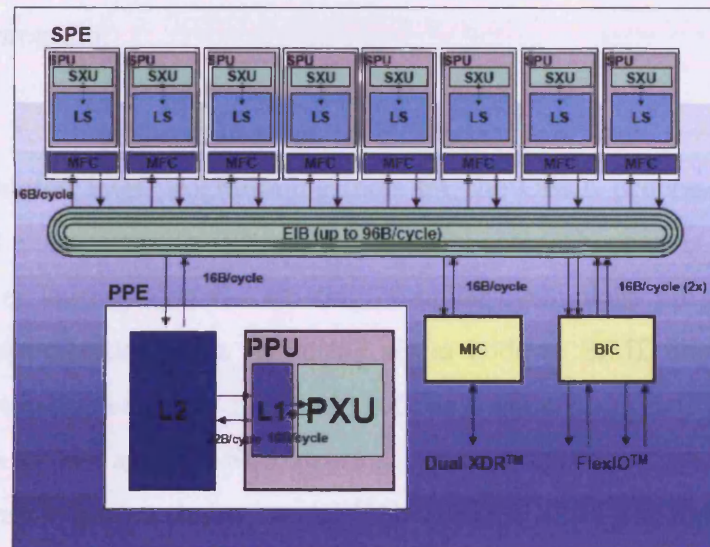
One of the most novel developments using FGPA based technology was the Convey HC-1 Computer [41]. The Convey computer integrates a FPGA based processor with a standard Intel 64 processor. The FPGA and the Intel processor share memory and can be dynamically reloaded with different instruction sets, dubbed "personalities". These personalities allow the relatively rapid addition of new instruction sets or machine features which are tailored to specific applications. These specialised instruction sets allow efficient acceleration of applications on the system as they are able to leverage instructions or machine features that have been specifically developed to allow the acceleration of a particular algorithm or class of algorithms.

Despite all of these different programming languages and libraries that are available, no one language has emerged as dominant in the field. The only widespread consensus is that some High Level language is needed and the choice is often largely dependent on developer preference. This diversity of programming languages available for FPGAs has not, however, hampered the work of porting applications to this device and all of the languages have had

applications successfully ported to them.

2.5 IBM Cell Broadband Engine

The IBM Cell Broadband Engine (CELL/B.E.) processor was a joint development by Sony, Toshiba and IBM. The CELL itself is a high-performance, multi-core processor with a custom system-on-a-chip (SoC) implementation [121].



Source: M. Gschwind et al., Hot Chips-17, August 2005

Figure 2.9: The CELL/B.E. Architecture adopted from [119].

Figure 2.9 shows the structure of the IBM CELL processor. The CELL chip itself consists of an IBM 64-bit Power Architecture core, the Power Processing Element, and eight specialised co-processors based on a single-instruction multiple-data (SIMD) architecture, these are the Synergistic Processing Units (SPE), which provide the parallel processing power of the chip.

The CELL/B.E. architecture is programmed using either a modified version of the GCC toolchain or the IBM XL C/C++ compiler toolchains [42]. Both of

these compilers provide cross compilation from the development architecture to the PowerPC Architecture, for programming on the PPE, or to the custom SPE Architecture. The SPE components for both compilers support a set of extensions for SPE programming.

The executables generated by these compilers contain both the instructions for the PPE and instructions for the SPE units. Each program will contain one set of instructions for execution on PPE, but there may be several sets of instructions for the SPEs, allowing the execution of multiple kernels during the runtime of the overall program.

In addition to these programming tools, there have been several efforts to implement higher level programming tools for the CELL processor. IBM itself has released their compiler system, which has become known as the Octopiler. This compiler leverages on the existing compiler technology employed by IBM to provide optimisation when executing scalar code in SIMD units, and allows the auto generation of vector instructions from a scalar source, this is performed by a process known as Auto-SIMDization. The Octopiler also provides support for the OpenMP programming model [51]. Bellens et al [19] have developed a system called CellSS operating at an even higher level of abstraction. CellSS is not limited to the CELL architecture, although it is the architecture supported by their prototype. CellSS uses a code commenting style, similar to OpenMP, to identify tasks within a sequential program. The compiler uses these annotations to separate the main program code, which will run in the PPE, and the task code, which will run in the SPE. When the subsequent program is executed, the annotated tasks are submitted for execution. The execution of these tasks is controlled by a task list. When all data-dependencies for a task are met, it is added to the task list. The runtime system monitors this task list and then matches tasks in the list to available resources (The SPE processors in the case

of the CELL).

The majority of success for the CELL processor has come from its integration into the Sony PS3™(PlayStation 3) and this console is currently the most accessible way for users to gain access to CELL technology. Recently, work has even been done utilising PS3's™as scientific computing machines, essentially becoming acceleration devices in their own right. This has even been extended as far as the creation of PlayStation clusters. [80].

2.6 Computational Libraries

The use of computational libraries is a mainstay of development for standard sequential CPUs and multi-core architectures. Many such libraries have been developed such as LAPACK(Linear Algebra PAckage), FFTW and many offerings from the Numerical Algorithms Group(NAG) to name but a few. These libraries allow developers to leverage expertly optimised algorithms in order to achieve the best possible performance for common computational tasks. One common approach that may be taken by developers is the investigation of three possible options for optimising their code:

- Automatic optimisations provided by compilers or similar tools.
- Re-factoring of code.
- Identifying parts of their code that can be replaced by calls to a numerical library.

A similar approach can also be taken by developers utilising acceleration devices, with the main difference being that when considering an acceleration device, the

computational libraries are generally device specific, allowing them to optimally utilise the varying architectures of acceleration devices. Although application acceleration is still a rapidly developing technology, many algorithms have been ported and optimised for use on these devices. This section will analyse existing work in this field that is relevant to the remainder of this thesis.

When considering computational libraries that have been developed for application acceleration devices, the libraries developed and shipped with the devices are often the first encountered by developers.

NVIDIA provide developers with a set of libraries including Basic Linear Algebra (BLAS), Fast Fourier Transforms (FFT), sparse matrix routed and random number generation [99] and these libraries in certain circumstance can provide performance approaching the device's maximum level [107]. ClearSpeed also provides users with a similar extensive set of libraries [35] and IBM provide, among others, a BLAS library [4]. AMD also provides developers with their Core Math Library [12] which provides implementations of BLAS, FFT, LAPACK and random number generation algorithms.

In addition to the development of libraries by the device manufacturers, third party libraries have also been developed. MAGMA (Matrix Algebra on GPU and Multi-core Architectures) is one of these libraries. The MAGMA library provides wide variety of routines including: GEMM, Linear Solvers and LU, QR and Cholesky factorizations [6]. Recently Nath et al have also conducted other work on optimisation of the MAGMA GEMM algorithm on FERMI GPUs [97]. In this paper, to attempt to further improve the performance of the GEMM algorithm within MAGMA, the authors explore using several techniques, including pointer redirection to overcome the performance dips often seen in cases where the matrix size is not divisible by the block size (selected based on GPU characteristics) used to sub-divide the matrix.

Another library that has been developed is FLAGON. FLAGON is a library for programming NVIDIA GPUs using the Fortran 95 programming language[2]. The FLAGON library, instead of implementing all its own functionality, leverages on that provided by CUFFT, CUBLAS and CUDPP(CUDA Data Parallel Primitives) libraries.

The final linear algebra library that has been examined is CULA[68]. The approach taken by the developers of CULA is to attempt to abstract away the GPU specifics. They provide implementations of many common LAPACK functions that developers can utilize via an API that hides all GPU implementation specific details from the programmer. This level of abstraction allows developers to rapidly develop GPU accelerated linear applications, and performance figures quoted in[68] show that CULA running on a NVIDIA Tesla C1060 provides between 1.75x and 4x speedup when compared to Intel's MKL library, running on a Intel Core i7 920.

Another related library that is now available is Turbostream[8][22]. Turbostream is a computational fluid dynamics library that supports both GPUs and multi core CPUs. The Turbostream library utilises the SBLOCK solver which is an optimised version of the older TBLOCK solver which has been adapted to allow fine-grained parallelism.

Another area in which there has been much development, especially in regards to the GPU, is Molecular Modelling. One of the most widely used Molecular Modelling packages, while not strictly a library, is NAMD. NAMD now has extensive support for GPUs and can also utilise GPU clusters. Performance figures have shown that benchmarking NAMD on a 60 GPU cluster provided performance equivalent to that of 330 CPU cores[125]. Another similar success story was the addition of GPU support to the Folding@Home project, with the GPU implementation of Amber 9 providing speed-up factors of well over 100

compared to a single core CPU. Currently the vast majority of Folding@Home computing power is being provided by GPUs[125].

Although only a brief overview of applicable libraries has been provided in this section, this will be expanded upon in Chapter 6 where applications and libraries will be utilised as comparison for the system described in this thesis.

2.7 Existing Application Acceleration Porting Methods

As has been shown in the previous sections, there are currently several different types of Application Acceleration devices currently in use. Some have evolved to become acceleration devices from their use in a more specialised role i.e. FGPAs and GPUs. Some have been specifically designed as acceleration devices.

This competition forces the manufacturers to constantly improve the performance of their devices and provide better tooling. However, this situation presents several challenges to today's developers:

1. Selection of an appropriate device and/or programming language.
2. Difficulty in programming the device.
3. Before the work of porting an application begins it is difficult to predict the performance gain that can be achieved, or the suitability of the application for the selected acceleration device.
4. Once ported, the application is locked in to a particular device or manufacturer.

This often means that re-porting is needed when new generations of devices are released, or when a device offering better performance becomes available.

Once all these problems have been solved, it could theoretically be possible to develop a heterogeneous system for application acceleration. Such a system, illustrated in Figure 2.10 would be built from application accelerators and standard CPUs, allowing it to provide efficient execution for all application types. This ideal system would also have sufficient abstraction to allow users to simply compile and run their applications, without having to know what device their code will execute on. In short such a system would have to contain the following functionality:

- The ability to port code with no user intervention.
- The ability to select the device to execute the application.
- The ability to target all devices.
- The ability to operate on all known applications. This ideal system can be used as a target for work in the field of application acceleration, and has been the motivation for academic work looking to solve the problems illustrated above.

The research outlined in this section focuses on increasing the available level of abstraction to encourage the widespread adoption of application accelerators and take the use of these devices out of the computer science domain and into the application domain.

Howes et al., has made several contributions to the field [67], using ASC (A Stream Compiler), which is a class library for C++. The authors developed a unified source description providing cross compatibility between the FPGA,

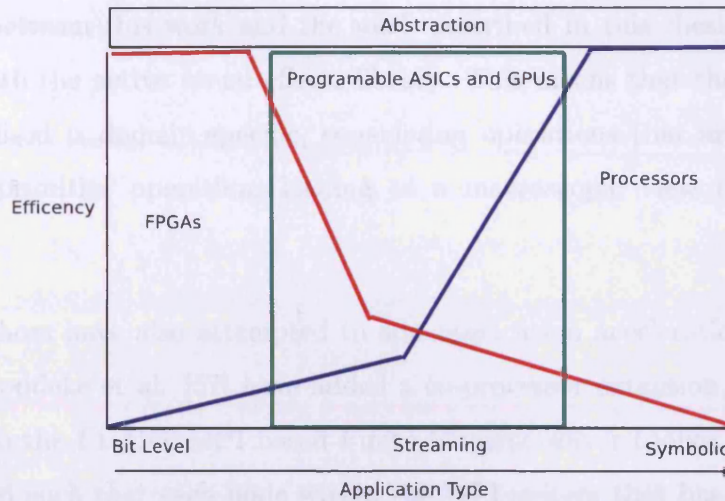


Figure 2.10: An Ideal Abstraction for Application Accelerators adopted from [29]

GPU and Vector units on the PS2TM. This work then enabled performance comparisons to be performed comparing the three devices running a Monte Carlo code, an FFT and a weighted sum application. The FPGA performed best out of all the devices for the FFT(although not outperforming the CPU). The GPU far outperformed all the other devices when running the Monte Carlo simulation and the PS2 outperformed the others when running the weighted sum. This work illustrates that selecting the most appropriate architecture for the application is important to achieve the best performance, however, it does not provide any means of using the performance data collected to select a device for the future executions of applications.

In the same research group as Howes, Cornwall et al. have developed a source to source compiler [40]. This compiler takes as input C++ code using an active visual effects library and produces CUDA output. The authors then continue to outline a series of useful domain specific optimisations that can be performed on the output source. The experiments conducted by the authors show that the project was a success, delivering speed ups of between 1.3x and 6.6x. The main

difference between this work and the work described in this thesis is the tight coupling with the active visual effects library. This means that the compilation process utilised is domain specific, considering operations that are part of the library as primitive operations leading to a macroscopic view of the overall application.

Several authors have also attempted to add application acceleration to existing systems. Goddeke et al. [57] have added a co-processor extension, with a GPU back-end, to the FEAST MPI based Finite Element solver toolkit. This toolkit is structured such that each node within the MPI system that has a compatible GPU, uses the FEAST-GPU extension to enable it to use the GPU as a co-processor for the computation being performed on that node. In addition to work using MPI, work has also been conducted using OpenMP. Lee et al [81] has developed an OpenMP to GPGPU compiler framework, allowing the translation of shared memory OpenMP programs to CUDA-based GPGPU programs.

Kunzman [79] has proposed a system, using the Charm++ programming model. The authors expand the Charm++ model with accelerated entry methods, accelerated blocks and an abstraction for SIMD instructions. These extensions allow programmers to develop, using the Charm++ system, applications that make use of the CELL. B.E. hardware, without the programmer having to be aware of the underlying hardware programming model. This system also provides automatic CPU fallback when a CELL device is not available.

Finally, Garg et al [54] have developed a system to compile Python code to a hybrid execution environment consisting of a CPU and an AMD GPU. This system is based on a series of annotations to determine if loops are executable on the GPU and, if they are executable, are they profitable for execution on the GPU. Once the user has identified these loops, several software tools are then used to provide a compilation framework. This framework first converts the

Python code into C/C++ using unPython. Then, loops that are not profitable for execution on the GPU are converted into OpenMP loops, while loops that are suitable for the GPU are converted to code that can be compiled using the AMD Compute Abstraction layer, using software that the authors have coined jit4GPU. This paper is interesting because it is the first paper that has been seen to allow the programmer to differentiate between loops that are parallel and those that are parallel and suitable for execution on the GPU. This is important in the case of small loops, which, while they may be easy to parallelise, contain so little computation that the speedup achieved by executing on the GPU is not sufficient to overcome to the overheads of moving data from the host's main memory to the device.

The work that has been outlined above has very successfully provided abstraction between several different application acceleration devices and this in turn has allowed useful performance comparisons to take place. However, all these systems fail to tackle several key problems:

- They all require the manual selection of the acceleration device.
- Few systems have achieved support for a wide cross section of acceleration devices.

In addition to these academic efforts there have been several products developed within industry to facilitate the use of application acceleration devices. These products are discussed in the following sections.

2.7.1 Portland Group Accelerator Compiler

The Portland Group Accelerator Compiler [134] is currently one of the most advanced systems for developing applications for the NVIDIA GPU. The Portland Accelerator compiler allows developers to compile from C or Fortran to CUDA with the addition of a set of compiler directives defined by PGI [62]. An example of such a compiler directive is shown in Listing 2.7.

Listing 2.7: PGI Accelerator Compiler Example adopted from [134]

```
int main( int argc , char* argv [] )
{
    int n;
    float *restrict a;
    float *restrict r;
    n = 100000;
    a = (float*)malloc(n*sizeof(float));
    r = (float*)malloc(n*sizeof(float));
    e = (float*)malloc(n*sizeof(float));
    for( i = 0; i < n; ++i ) a[i] = (float)(i+1);

    #pragma acc region
    {
        for( i = 0; i < n; ++i ) r[i] = a[i]*2.0f
        ;
    }
    return 0;
}
```

From the example shown in Listing 2.7 it can be seen that the only additions are the *#pragma acc region* and the *restrict* keyword. The addition of the compiler directive instructs the PGI compiler that the following code block is what should be accelerated. It is possible when defining these blocks, for the developer to specify additional options such as which data are copied to the device or define a set of conditions when the device should not be used and execution should

fallback to the CPU.

However, one of the most compelling features of the PGI compiler is its ability to auto-detect the majority of these additional options without the developer defining them. This includes the ability for the system to detect [134]:

- If the loop contains loop level dependencies preventing its acceleration. This is done by making the assumption, based on the use of the *restrict* keyword, that pointers point to different locations in memory.
- If nested loops need to be re-ordered and re-order them as appropriate
- If the loop is not parallelisable the compiler will attempt to detect why and provide feedback to the developer.
- Able to detect performance bottlenecks such as memory-stride or memory-alignment problems and report to the user.

In addition to these features, the PGI Accelerator compiler is able to perform optimisations on the code when it is ported to the GPU, the most notable of which is its ability to re-order nested loops and to vectorise parts of the accelerated code.

All of these features makes the PGI compiler one of the most advanced parallelising compilers currently available and it is an excellent tool for porting code to the NVIDIA GPU with reasonable automation.

2.7.2 Sieve Multicore Programming System

The Sieve Multicore Programming System has been developed by Codeplay based on the concept of a Sieve [120]:

- A sieve is a block of code contained within sieve { } markers and any functions that are marked with sieve.
- Inside a sieve, all side-effects are delayed until the end of a sieve
- Side effects are defined as modifications of data that are declared outside of the sieve.

The definition of a sieve given above, effectively allows the user to explicitly specify which sections of the program code the compiler can attempt to automatically parallelise. In effect the sieve block separates what is stored in the main memory of a many-core systems(outside of the sieve) and what would be stored in the local memory of a core in a many-core system(inside the sieve). An example of a sieve block is shown in Listing 2.8.

Listing 2.8: Sieve Example

```
sieve {  
    for (int i=0; i < n; i++ ) {  
        c[i]=a[i]*b[i];  
    }  
}
```

The sieve system developed by Codeplay consists of an extension to a C++ compiler, which extracts sieve blocks, performs some automatic parallelisation on them and then compiles them according to the sieve rules. The output of compilation with the sieve system is a set of C files, one for the control processor and one for each of the cores within the many-core system. Different back-ends have then been developed based on this output, including a CELL back-end. An OpenCL back-end is also advertised as being developed.

2.7.3 HMPP

HMPP, is a Heterogeneous Multi-Core Parallel Programming environment, which aims to allow programming of application accelerators at a higher level of abstraction and allow cross compatibility between multiple devices [49]. HMPP utilises source annotations entered by the programmer, to select codelets for acceleration. An example of these annotations is shown in Listing 2.9 for the codelet code and Listing 2.10 for the call site of the codelet.

The HMPP system consists of a preprocessor and a runtime system. The HMPP pre-processor will extract codelets based on the programmer's annotations and then compile these codelets, using vendor tools, to all available back-ends creating a library of codelets that can be selected for use. The HMPP pre-processor also inserts calls to the HMPP API at the call site to invoke the codelet.

The HMPP runtime, is responsible for loading the required device code based on what device is present or providing CPU fallback if there is no device is available or if there is no compiled codelet for the available devices.

Listing 2.9: HMPP Example Codelet Definition

```
#pragma hmpp mul codelet output=c
void mul(int n, int * a, int * b, int * c) {
    for (int i=0; i < n; i++ ) {
        c[i]=a[i]*b[i];
    }
}
```

Currently the HMPP system supports C and Fortran front ends and back-ends for CUDA, OpenCL and the AMD Compute Abstraction Layer(CAL).

Listing 2.10: HMPP Codelet Callsite

```
//item-wise multiplication of two 100 elements a and b.  
Output stored in c  
#pragma hmpp mul callsite  
mul(100,a,b,c);
```

2.7.4 Rapidmind and Ct: C for Throughput Computing

Rapidmind Inc. was, until its acquisition by Intel in August 2009, a software company providing a development environment which provided abstraction between GPUs, Multi-Core CPUs and CELL. B.E.

The Rapidmind system is programmable using the C++ language, with a series of macro, data and API type additions. An example of a Rapidmind program is shown in Listing 2.11. It can be seen from this example that Rapidmind allows the programmer to define a function to execute over an array of values in parallel. The Rapidmind programming system takes care of the generation of code to manage the data movement back and forward to the target device.

The Rapidmind compiler, which is essentially a preprocessing system for a standard C++ compiler and device vendor tools, currently supports outputting to CUDA, AMD CAL and C++ for the CELL. B.E.

Rapidmind publicity, even though licenses are still being sold, has largely disappeared following its acquisition by Intel.

Intel have, however, developed their own parallel processing tools. These are called Ct(C for Throughput computing) [56] and they are currently focused on the Intel's Multi-Core architectures, although it is assumed, but not confirmed, that at some point the Rapidmind product will be merged into Ct. This will

Listing 2.11: Rapidmind Example

```
//Element-wise Multiplication of two Arrays A and B
int main() {

    MUL=RMBEGIN {
        In<Value1i> a;
        In<Value1i> b;
        Out<Value1i> c;

        c=a*b;
    } RMEND;

    Array<1, Value1i> A;
    Array<1, Value1i> B;
    Array<1, Value1i> C;

    //execute the program
    C=MUL(A,B);
}
```

provide Intel with a system that enables cross compatibility between Multi-Core processors and Acceleration devices.

Ct is a C++ extension which provides parallel programmability to the programmer by allowing the use of a series of managed parallel datatypes, such as the TVEC(a managed parallel vector). Along with these parallel data types the system also provides associated element-wise and collective communication operators for use with the new data types [56]. An example of the use of the element-wise multiplication operator is shown in Listing 2.12, this operation can be used by the programmer independently of the size or shape of the two vectors.

Listing 2.12: Ct Code Example

```
//Element-wise Multiplication of two Vectors A and B  
TVEC<F32> A;  
TVEC<F32> B;  
TVEC<F32> C= A*B;
```

2.7.5 Peakstream

Peakstream [112], along with Rapidmind, was one of the earliest commercial application acceleration software development packages and the two products were in heavy competition. The Peakstream product ceased development in 2007 when the company was acquired by Google. It is mentioned here for completeness only as information on Peakstream products has all but disappeared from the public domain.

2.7.6 Evaluation of Existing Methodologies

The previous sections have examined many of the academic and industrial efforts to solve the obstacles to the adoption of application acceleration outlined in Section 2.7. An overview of the current solutions are shown in Table 2.1 and a complete time-line of all released solutions (i.e. those that have been made available for wider use) are shown in Figure 2.11, this figure also shows, for comparison, some of the other key software tools that have been discussed in this chapter.

This table shows that, while several systems have made efforts to solve the problems of programmability and device lock-in, there is still no solution that provides total coverage and all solutions still require some form of user annotation or make use of a custom API.

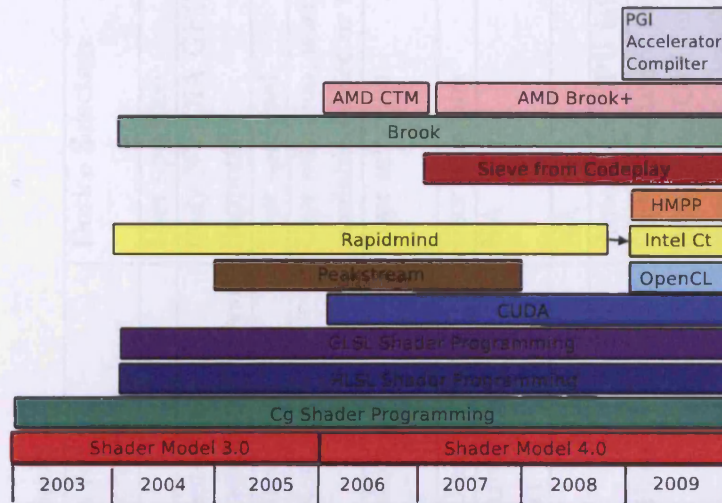


Figure 2.11: A time line of software released to facilitate development on Application Accelerators

However, there are, to my knowledge, no solutions currently available for the problems of device selection and performance prediction. At the moment there is no system able to give performance estimates prior to porting (with the exception of [66] specifically for the FPGA). There is also no system able to perform device selection for the programmer, a tool which will become increasingly desirable as the availability and diversity of acceleration devices increase and developers are given far wider choices when selecting a device and programming language, a choice that many will be ill-equipped to make.

Table 2.1: Comparison of Existing Porting Methods.

System Name	Devices Supported	Programming Language	Modifications to Programming Methodology	Device Selection
OpenCL	AMD/NVIDIA GPU	C	Custom API	User specified
PGI Accelerator Compiler	NVIDIA GPU	C, Fortran	Addition of notation with loop dependency analysis	Only NVIDIA GPU Supported
Sieve from Codeplay	CELL, OpenCL	C++	Addition of Sieve keyword	User specified
HMPP	AMD/NVIDIA GPU OpenCL + CPU	C, Fortran	Addition of Notation	User selection, selection of first available device or CPU fallback
Rapidmind	GPU, CELL	C++	Addition of Macros and API calls	User selection
Howes et al [67]	FPGA, GPU	C++	Addition of Class Library	User selection
Cornwall et al [40]	NVIDIA GPU	C++	Use of an Active Visual Effects Library	NA
OpenMP to GPGPU [81]	NVIDIA GPU	C/C++	OpenMP	NA
Kunzman et al [79]	CELL	C++	Charm++	Automatic CPU fallback
Garg et al [54]	AMD GPU	Python	Code annotations	Code that is not explicitly flagged for GPU execution is run using OpenMP

2.8 Chapter Summary

This chapter has provided an overview of the four most common application acceleration devices: The GPU, ClearSpeed, CELL and the FPGA. It has also outlined the various programming methods used to develop applications for these devices and has discussed the current state of the art developments in terms of the computational libraries that have been developed for these devices.

Finally I have analysed the current solutions that have been developed to attempt to break down the barriers to adoption of application accelerators outside of the Computer Science domain. A comparison of these solutions has been undertaken and the results are shown in Table 2.1, this analysis has identified several key problems that have not yet been satisfactorily solved.

The analysis conducted has shown that, compared to the ideal system shown in Figure 2.10, there are several areas where existing work is currently lacking, the key missing area of work being the lack of the ability of systems to select a device without human interaction.

In relation to other requirements of such an ideal system, several systems have increased the level of abstraction to reduce the required level of user intervention. However, these systems all require the programmer to learn a new programming language, API, or annotation style. Secondly, no systems have yet achieved complete coverage of the breadth of acceleration devices, although this is planned for systems such as HMPP and OpenCL. Furthermore, no software tools have attempted to solve the problem of matching an application to a device, or deciding if an application will give worthwhile performance improvement when ported. This is a major obstacle, as presently the time taken to port an application can be large, especially to a new user of acceleration devices, and the lack of

certainty of outcome will often discourage users from investing time and money in the technology.

These key problems currently form major obstacles to the widespread adoption of application acceleration devices and provide fertile ground for Computer Science research effort.

Chapter 3

Overview of the Application Porting System

3.1 Introduction

This chapter will outline the overall architecture of the system that had been developed in order to test the hypothesis described in Chapter 1.

From this hypothesis, and from the background research, it can be seen that the system must perform the following tasks:

1. Selection of the most appropriate device for the application,
2. Porting the source code to the device's programming method/API,
3. Compiling the ported code using the device's tool chain,
4. Using the performance results gathered from executing the ported code to modify the process of matching future applications to devices.

In order to construct this system, the initial starting point is the standard compile and execute model used by virtually all computer systems. Figure 3.1 shows at

the very highest level, the design of the system. The user will provide source code and data to the system, which will then return an executable to the user.

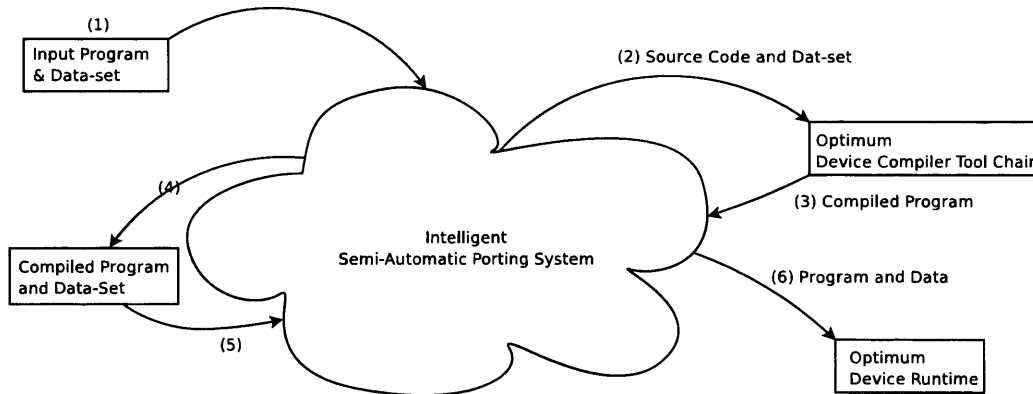


Figure 3.1: High level design of the Porting system

During this compilation phase we know, from our system's requirements, that the system must perform the following tasks:

1. Selection of an appropriate device.
2. Porting of the source code to the device's programming method/API.
3. Compilation using the device's tool chain.

During the execution phase, the executable, along with the input dataset, is passed back to the system. The system then executes it on the device for which it was compiled, returning the results to the user. It should be noted that, for performance reasons, the collection of performance data does not occur as the "production" version of the code is executing, but instead will happen as a background task.

The remainder of this chapter will now discuss in more detail the architecture of the Porting System.

3.2 Overall System Architecture

The overall architecture of the system will draw inspiration from several existing types of software.

Compilers

As mentioned in the previous section, the porting system will follow the standard compile/execute model. Secondly, as the porting system is a source to source compiler we can apply the same two stages that all standard compilers employ: Analysis and Synthesis [69]. This means that the porting system shall consist of a front end, which analyses code, and a back end, which synthesises code into the desired output format.

Web Services

One of the main goals in the development of the system is to enable the selection of the most appropriate device for the application being executed. This means that the system must have access to a variety of acceleration devices. This was the main motivation behind making the system distributed. If the system was not distributed, then only the acceleration devices connected locally to the user's computer would be accessible. This would immediately place a limit on the number of devices a system could support - the number of free expansion slots available. Additionally, currently many acceleration devices are purchased as shared resources, and so are not connected to an individual's machine, but rather to a central server. In order to leverage on these devices the system must be

distributed. These reasons make it highly beneficial that the system is of a distributed nature.

The decision to make the system distributed introduces several new items of functionality that the system must provide:

- Ability to locate devices.
- Ability for distributed components to communicate.

Taking all these requirements into consideration the system architecture was refined and the new, more detailed, architecture is shown in Figure 3.2. The diagram shows the system's three modes of operation (with solid lines representing network traffic), and for each mode of operation the order of operations are shown:

1. Compilation (Shown in red).
2. Execution (Shown in blue).
3. Collection of Performance Data (Shown in green).

The system is divided into four components. These components, with the exception of the client which is installed on the users machine, are all presented as web services. The decision to use web services, was taken purely to allow this work to leverage the pre-defined communications protocols for web services. Each of the four components are now discussed in further detail below.

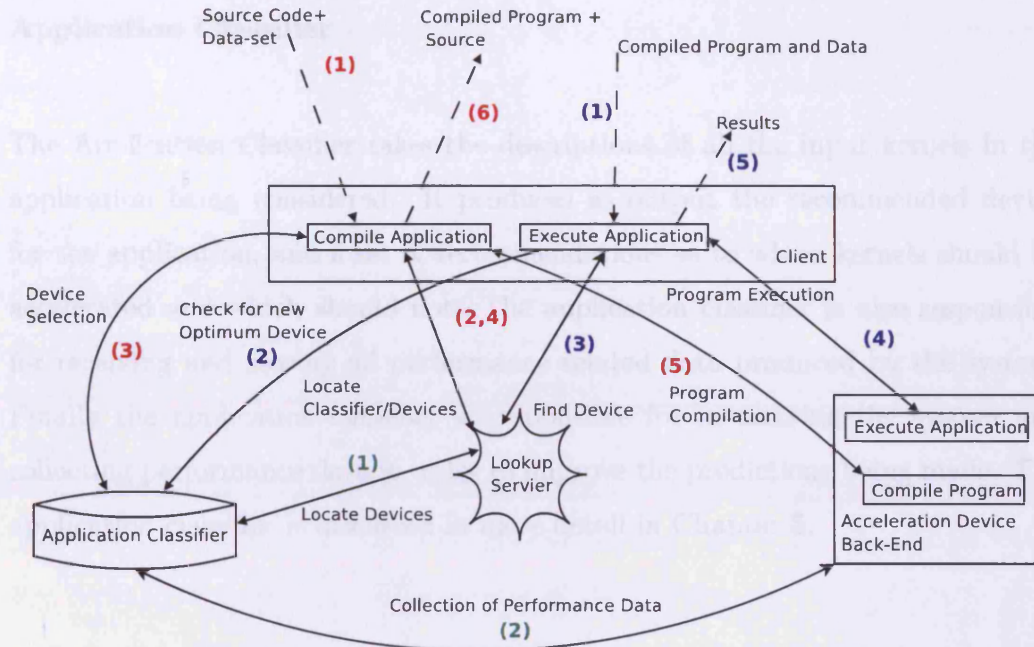


Figure 3.2: The Application Porting system

Client

The client is the only component of the system that is installed on the user’s computer and takes as input suitable source code containing at least some parallel elements. The client handles front end parsing of the input source and acts as a driver for the porting process. The majority of the client’s processing is related to the extraction and analysis of the input code. It takes the input and from it extracts small chunks of code that are candidates for acceleration, these chunks of code are known as “kernels”. The client is discussed in more detail in Section 3.3.

Application Classifier

The Application Classifier takes the descriptions of all the input kernels in the application being considered. It produces as output the recommended device for the application, and a set of recommendations as to which kernels should be accelerated and which should not. The application classifier is also responsible for receiving and storing all performance related data produced by the system. Finally the application classifier is responsible for monitoring the system and collecting performance data in order to improve the predictions being made. The application classifier is discussed in more detail in Chapter 5.

Accelerator Back-End

The Accelerator Back-Ends take as input: the host source code, the set of kernel descriptions and associated kernel source code as produced by the client and the recommendations produced by the application classifier. The back-end will then port the input code to the appropriate target language as required by the device and then compile it using the device's tool chain. The back-end is also responsible for gathering performance data from the applications as they execute and passing it to the Application Classifier. The Accelerator Back-Ends are discussed in more detail in Chapter 4.

Lookup Server

The lookup server is a Universal Description Discovery and Integration (UDDI) server, running Apache jUDDI. The UDDI server sees the distributed components of the porting system as a set of "Accelerator" services and a single "Classifier"

service. The server maintains a list of the locations(IP addresses) of all of these services which it provides, upon request, to the other components in the system.

These four components all encapsulate the key functionality of the system. The accelerator back-ends provide the ability to generate of device specific code, which is discussed in Chapter 4. The Application Classifier, described in Chapter 5, provides the functionality related to the self-modification of the system's classification model, the ability of the system to expand itself and the core functionality of device selection.

The remainder of this chapter will consider the functionality of the client, which is a critical part of the overall operation of the system.

3.3 Overview of System Client

The system client is the driver behind the compilation and execution processes for the porting system, integrating the distributed components into one cohesive system. The client is also responsible for performing the code analysis tasks performed by the front end of a compiler. The client itself operates in two different modes; compilation and execution.

The overall structure of the compiler client is shown in Figure 3.3. The communication tasks that are performed by the client are shown with dotted lines, while each main analysis task of the client is shown in a box. Each of these tasks are discussed in more detail below, while the execution client is discussed in Section 3.8.

The current version of the client constructed for this work supports a single input language: ANSI C with a few restrictions:

- Unstructured jumps, i.e. *goto* statements, are not permitted,
- The current version can only parse array accesses using the indexing operators `[]` and not pointer arithmetic.
- All memory allocations must be visible within the bounds of the code provided to the client.
- The current version assumes that all pointers to arrays are non overlapping. i.e. if *a* and *b* both point to arrays, it is assumed that they point to different arrays and neither points to a subset of the other.

The approach taken, however, is generic and could be extended to support other imperative languages such as Fortran 90.

3.4 Source Parsing and Validation

The first analysis stage of the client performs parsing and validation of the input source code. This stage of the client takes as input a directory containing the input program and then performs the following tasks:

1. Locate all necessary code files.
2. From the code files, locate all required header files, ignoring those that are part of the operating system libraries.
3. Pre-process all source files, with the exception of `#include` directives.
4. Parse each method within the code, starting with `main`, using the CTOOL (<http://ctool.sourceforge.net>) library.

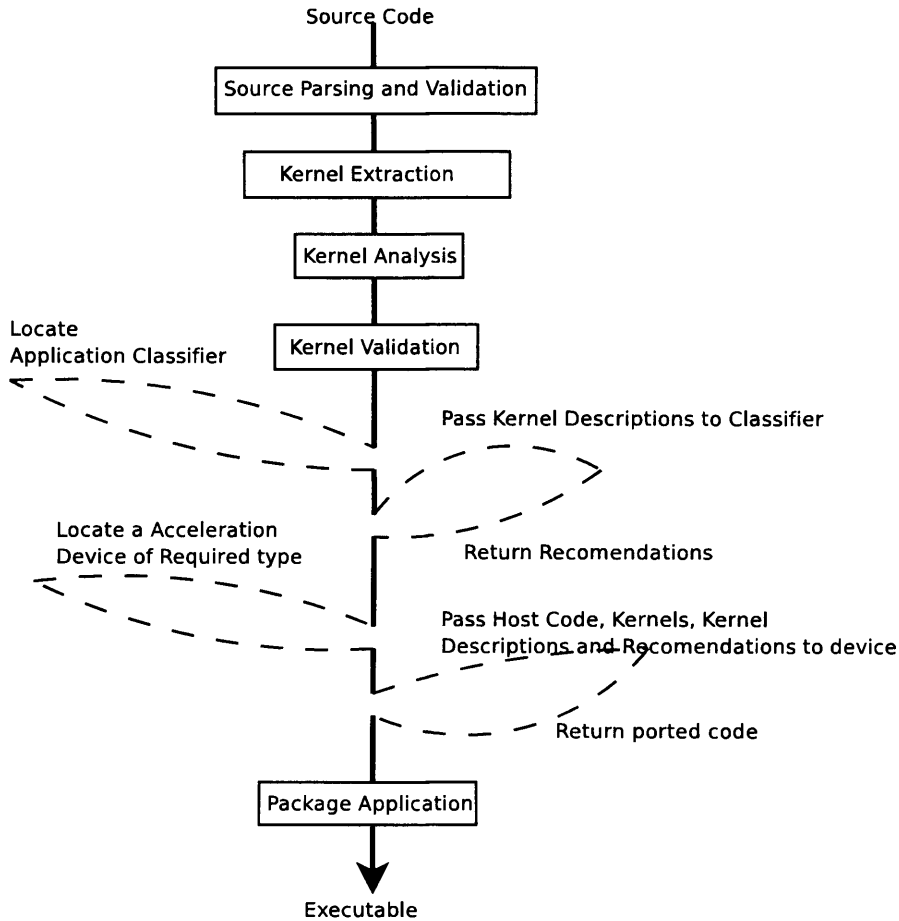


Figure 3.3: Client in the Compiler Mode

This stage will result in (assuming the input code is valid) a set of Abstract Syntax Trees, generated by CTOOL, for all the methods within the input application. For the purposes of identification each abstract syntax tree is associated with its function definition. These Abstract Syntax Trees are then passed to the next stage: Kernel Extraction.

3.5 Kernel Extraction

The extraction of kernels from the input source code centres around finding the loop level parallelism within the code. This is done automatically in several steps:

Constructing a Control Flow Graph

The first step is to construct a control flow graph for each abstract syntax tree produced by the previous stage. A control flow graph consists of nodes, representing basic blocks within the code and edges, representing the flow of control. In addition to the standard nodes generated by flow graph analysis, additional nodes, which contain no code, are added to mark the entry and exit of branches [27], this is to assist the later stages of the client in processing the graph.

Basic Block: A basic block is a sequence of consecutive statements in which the flow of control enters and leaves at the end without halt or the possibility of branching except at the end [10].

An example of a control flow graph for a piece of code is shown in Figure 3.4 and the code that the graph represents is shown in Listing 3.1.

Kernel Formation and Separation

Once a control flow graph has been constructed the client will extract an exhaustive set of all possible kernels from the code. This in essence consists of all natural loops within the code. It should be noted that many natural loops are unsuitable for execution on any acceleration device e.g. because input/output

Listing 3.1: Code Example for Control Flow Graph

```
b=100;
while (a < b) {
    b--;
    if (a < b) {
        a=a+1;
    } else {
        a=a-1;
    }
    a=a+10;
}
b=0;
```

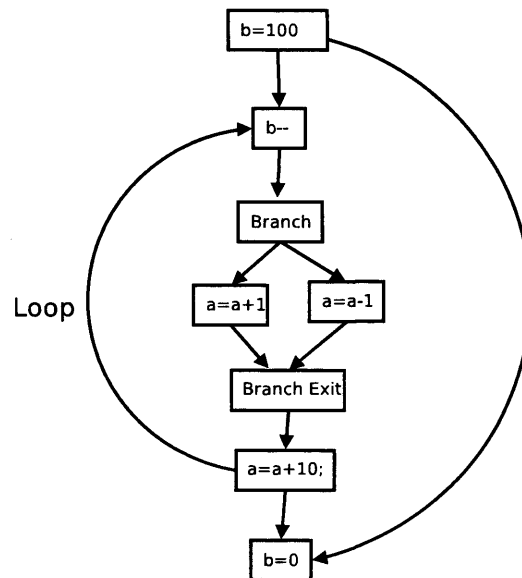


Figure 3.4: A Control Flow Graph

takes place within the kernel or the loop that formed the kernel may be non-deterministic. These kernels will be filtered out by later stages of the system.

In order to define what a natural loop is we must first define the concept of one basic block dominating another.

Dominators: A node d dominates node n , if every path from the initial node of the flow graph to n goes through d [10].

Loop Entry: A loop entry node is single entry point into the loop. This entry point dominates all nodes in the loop and there must be at least one path through the loop back to the entry node [10].

Natural Loop: A natural loop is defined as a collection of nodes, all dominated by the loop entry node and all strongly connected such that from any node in the loop to any other, there is a path of length one or more, wholly within the loop [10].

Back-Edge: An edge that connects a node to an ancestor within a tree[10].

Reverse Postorder Traversal: When traversing a tree, each node is visited before all of its successor nodes, except when the successor is reached by a back edge)[10].

As the first step in kernel formation, the client will traverse through the data flow graph for each function, locating loops and labelling each node within a loop. The algorithm used is based on that outlined by Alfred [10] and later used by Callahan [27] and is shown in Listing 3.2. The algorithm loops through each node in the control flow graph in reverse postorder, and for each node performs the following actions:

- If a node n is a loop entry node, then it is part of loop n .
- Determines if the node is a member of any other loops by checking if it is dominated by the loop entry nodes of any loops to which its predecessors belong to.

The end result of this is that each node in the flow graph is annotated with the id of each loop that it is part of. An example of a fully labelled control flow graph is shown in Figure 3.5, with associated source code shown in Listing 3.3. This figure shows how nested loops are handled and that it is quite correct for an inner loop to be shown as part of multiple loops.

Listing 3.2: Algorithm for Loop Identification

```

Inputs :
G= Blocks from the Control Flow Graph ordered in reverse
    postorder

for all nodes n in G
  if n is a loop entry node
    Add the node ID of node n to list of loop IDs for
      node n
  end if

  P = All predecessor nodes to n
  for all p in P
    for all loop IDs i assigned to node p
      if loop entry block of i dominates n
        add i to list of loop IDs for node n
      end if
    end for
  end for

end for

```

With all loops now identified within the control flow graph, they can be separated

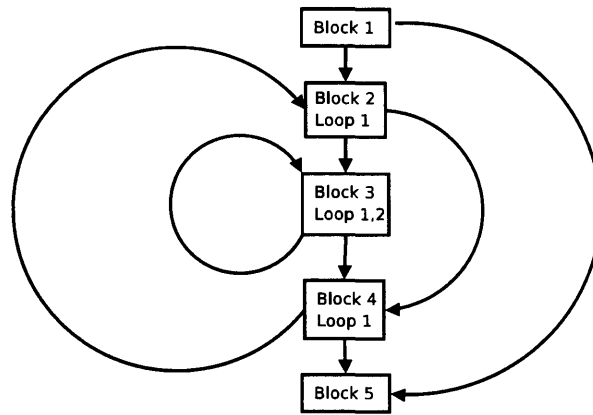


Figure 3.5: A Control Flow Graph, labelled with loop IDs

Listing 3.3: Source code that generated Figure 3.5

```

X=500;
for (int i=0; i < X; i++) {
  //Loop 1 Code
  for (int z=0; z < X*2; z++) {
    //Loop 2 Code
  }
  //Loop 1 Code
}
X=0;

```

into kernels. This is done by taking each loop, starting with the innermost, removing it from the control flow graph and placing into a new flow graph representing a kernel. A placeholder node is then placed into the original flow graph at the location the loop was previously. A separated version of Figure 3.5 is shown in Figure 3.6. It is important to note that in Figure 3.6 Kernel 2 is a sub-kernel of Kernel 1, as prior to processing, Loop 2 was an inner-loop of Loop 1.

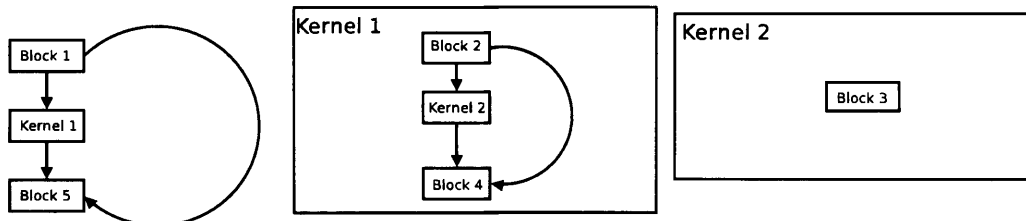


Figure 3.6: A Control Flow Graph, with Kernels Separated

3.6 Kernel Analysis

Once all kernels have been extracted from the application source code, each kernel is in turn analysed and information is extracted from its code. Some of this information is useful to the client for its processing, while some are metrics used in matching the application to an appropriate acceleration device.

The Kernel Analyser takes as input the control flow graph for the kernel and extracts the following information from it:

- The kernel ID.
- The kernels parent kernel (if it exists).
- A list of sub kernels.
- The source file, from which the kernel originated prior to processing.
- A list of variables written to and read from inside the kernel, and their sizes.
- List of array indexes used to access variables that are read from or written to.
- Details of the loop instruction that caused the kernel to form.
- List of functions called by the kernel.

For the sake of organisation, only the analysis related to the client's processing is discussed here. The extraction of the metrics used in the decision making process is discussed in Chapter 5.

Once the analysis of all candidate kernels is complete, the kernel analyser outputs, for each kernel, a Kernel Description File, an example of which is shown in Figure 3.7.

```
ACCESSMAP:a,i * k + y:b,j * k + y:c,x:a,i * k + y:b,j * k + y:
ACCESSMAPWRITE:cOut,x
BRANCHING:0
DATAIN:120020
DATAOUT:40000
FILENAME:src///dgemm.c
FUNCTIONS:
INTENSITY:14
ISSUBKERNEL:N
ITERATIONS:10000
KERNEL:3
LOOP:finite:for
LOOPCOND:x = 0:x < n * m:x++
LOOPCONTROL:x
PARENTKERNEL:-1
READVAR:int ,m:float [sizeof(float ) * m * k/sizeof(float )],
SOURCEFILE:src///dgemm.c
SUBKERNELS:4
WRITEVAR:float [sizeof(float ) * n * m/sizeof(float )],cOut
```

Figure 3.7: A Kernel Description File

3.7 Validation of Input Kernels

With the kernel processing now completed, the set of kernels within the program can be visualized as a kernel tree, such as the one shown in Figure 3.8. Validation is performed individually on each kernel within the tree and, as part of this process, each kernel is either passed or discarded based on a series of tests:

Filtering Kernels not executable on any device: This phase of kernel validation is performed on the system client prior to the selection of the target

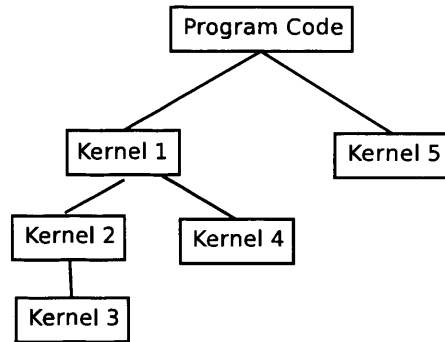


Figure 3.8: A Kernel Tree

device and consists of checking the kernel against one key requirement applicable to all devices:

- That the amount of iterations of the loop that formed the kernel is known before execution of the loop begins, i.e. it is a deterministic loop.

Filtering Kernels with loop dependencies: The second step is also performed by the system client, and its aim is to filter out some of the kernels with dependencies prior to device selection or porting.

Whilst much theoretical work has been conducted into array dependency analysis, such as the Omega Test by Pugh [117], the work in this thesis, however, takes a far more pragmatic approach. It was decided that an exhaustive but complex dependency checker is not required, as the aim of this phase of kernel validation is limited to reducing the number of candidate kernels, rather than the completely accurate elimination of all kernels with dependencies. This approach accepts that some kernels will be incorrectly selected for execution. However, the modular approach that has been used in constructing this component of the system allows for a more sophisticated approach to be added in the future.

Instead the client applies a few simple heuristics to eliminate the majority of

kernels with dependencies:

1. If a kernel writes to a single variable(i.e. not an array) that is then used outside the kernel, this creates a race condition, so disallow the kernel.
2. If any array writes are based on an index value that is a constant, then the kernel will be disqualified.
3. If any array writes are based on an index value that is not calculated within the kernel, then the kernel is disqualified.

It should be noted, that the result of validation for a kernel in these first two steps, does not affect the result for its parent kernel or sub-kernels. The reason for this is because that if a kernel has array dependencies, then it would function as a sub-kernel.

Filtering Kernels not executable on the chosen device: The final filtering process is performed on the device back-end components of the system and will eliminate from the list of candidate kernels, any kernels that, while they are generally valid for acceleration, are not valid for acceleration on the considered device.

This process will involve traversing the kernel tree in post-order, checking the following for each kernel:

- Remove kernels containing recursion if recursion is not supported.
- Remove kernels containing function calls to non user-defined functions that are not supported by the device.
- Remove kernels using data types that are not available on the device.

- Remove kernels using language features not supported by the device's programming method. i.e. structures.

This filtering works slightly differently to that discussed previously. When filtering based on these characteristics, it should be obvious that if a kernel contains, for example, recursion, then its parent kernel would also contain recursion and thus also be an invalid kernel. However, conversely, if a kernel contains recursion it does not necessarily mean that its sub-kernels are also invalid.

If a kernel fails any of the above tests then it is discarded at this stage. Kernels that are discarded because they are not executable on a specific device are merged back into the control flow graph from which they originated, this may either be as part of another kernel or as the main body of source code that is not to be accelerated. However, kernels that are discarded because they contain loop dependencies are flagged as containing loop dependencies but are not rolled back up. This is because a kernel with loop dependencies could still be a valid sub-kernel.

Once the kernel validation has been completed, the client outputs the processed application code ready to be passed to the Application Classifier and the appropriate back-end. The original application source code that is not part of a kernel is outputted to the *src* directory, maintaining its original file and directory structure. The control flow graph for each kernel is output as source code into a *kernel* directory and the kernel description files are output into a *kernelloader* directory. In all cases, when the output function of the compiler encounters our custom node that represents a kernel call then a *#include* directive is output pointing to the appropriate kernel loader. An example of output from the client is shown in Figure 3.9, this particular example has one source file and five kernels.

```
-- kernelloaders
|-- kernel0.c
|-- kernel1.c
|-- kernel2.c
|-- kernel3.c
|-- kernel4.c
|-- kernel5.c
|-- kernels
|-- kernel0.c
|-- kernel1.c
|-- kernel2.c
|-- kernel3.c
|-- kernel4.c
|-- kernel5.c
-- src
  -- src
    -- dgemm.c
```

Figure 3.9: Client Output

3.8 Application Packaging

Once the appropriate device has been selected, and the application code has been ported and then compiled by the device back-end, the complete set of source files, build scripts and the compiled executable are returned. As, depending on device and application, this may be a significant amount of files, the client will then archive them. This archive file is then bundled along with a small script, which is responsible for invoking the execution client. This structure enables the user to execute the compiled application from a single executable file, in the same manner to which they would execute any standard program. The only custom command line argument that must be passed to this executable specify the path to the needed data files. An overview of the execution client that is invoked is shown in Figure3.10.

3.9 Chapter Summary

This chapter has described the architecture of the Application Porting System that has been created. This system is distributed in nature, so that it may

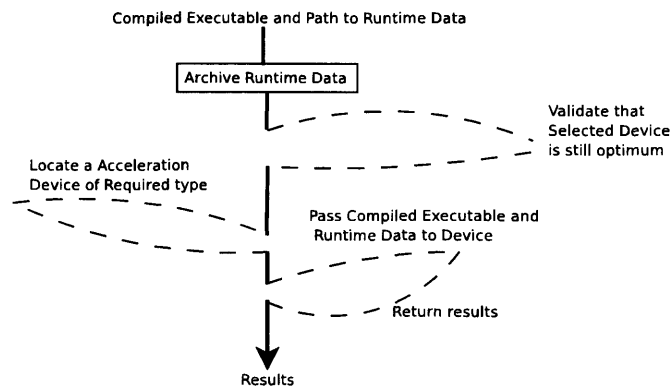


Figure 3.10: Client in the Execution Step

leverage on a wider range of acceleration devices than just those directly connected to the user's desktop computer.

This chapter has outlined the overall design of the system and it can be seen that the three key features of the system are:

- The ability of the system to filter the input application down to those sections of code that are executable on an acceleration device,
- Intelligently select an appropriate target device for these kernels,
- By collecting new performance data, modify its own decision making model so that the accuracy of the selections made will improve,
- Automatically port the selected kernels to the target device.

However, not all of the functionality of the system has yet been described. The code generation that the system undertakes is discussed further in Chapter 4 while the decision making, and collection of performance data is discussed in Chapter 5.

More specifically, this chapter has discussed in detail the components that make up the system client. These components, including the process of kernel extraction, kernel filtering and kernel analysis can be viewed as one large filtering process going from a full application down to “hotspots” in the application that are executable in parallel. These components have been developed as part of the production of this system and are built on a mixture of well-known compiler techniques and heuristics developed from the experience gained by working with application acceleration devices.

However, there are still ways in which these could be improved and it would have been preferable if the system could have been built by creating linkage between pieces of industry strength software, but, in many cases, such software is simply not available. Instead the system was developed in a modular fashion by combining a series of custom programs. This means that individual parts of the system can be improved in a modular fashion by replacing any of the components that have been developed so far.

One of the main components that could be improved are the heuristics used to filter out kernels with array dependencies; while the ones used in this chapter have functioned well for our testing, it is realised that these will not catch all cases of array dependencies. An ideal solution for this would be the implementation of the Omega Test [117] or using technologies similar to those used by the PGI Accelerator Compiler [134].

Chapter 4

Code Generation

4.1 Introduction

This chapter will show how the porting system fulfils one of its key requirements: the generation of device specific code. This functionality is largely contained within the Acceleration Back-End component of the system, discussed in Chapter 3.

The process that has been created for the generation of device specific code takes input consisting of the following:

- Application source code to be executed on the host.
- A set of kernel descriptions(Described in Chapter 3) each containing the characteristics of a candidate kernel.
- Kernel source code for each candidate kernel.

Each back-end will follow a process consisting of the following stages to generate code for its target device:

1. Selection of Kernels for Execution.

2. Porting.

While stage 1 must be customised for the each device, it will be, in essence, very similar for all devices. Stage 2, however, is radically different depending on the target programming language of the device being considered.

This chapter will examine in detail each of these stages, discussing stage 1 generically, and then stage 2 in detail for each device back-end that has been constructed.

The current back-ends that have been selected for construction are NVIDIA CUDA and C_N for ClearSpeed. This selection was made primarily due to limitations of available hardware, but the two back-ends that have been developed are sufficient to exercise the intellectually important elements of the system.

4.2 Kernel Selection

Once the initial kernel filtering process has removed all kernels that are presently incapable of being accelerated on the device, then the next stage is to select the kernels that will provide the best expected performance improvement. Figure 4.1 shows a kernel tree and an example of the selection of two kernels for execution. It should be noted that in Figure 4.1 as Kernel 2 has been selected for acceleration it also implies that Kernel 3 will also be accelerated.

The decision on which kernels are to be accelerated is taken by passing the kernel description of each kernel to the application classifier and, as each kernel description encapsulates all of its subkernels, a decision can be made based on the predicted performance returned from the classifier. The application classifier and

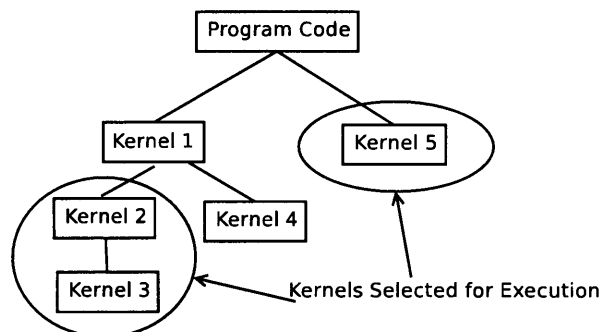


Figure 4.1: A Kernel Tree

this decision making process are discussed in more detail in Chapter 5. It should be noted, that at this point, since a target device has already been selected the decision that is made is simply whether to accelerate the kernel or not.

4.3 Porting to CUDA for the GPU

Once selected, each kernel that is to be accelerated is passed to the code generator. In the case of porting to CUDA, if the kernel being accelerated has subkernels then the subkernels are rolled up into the main kernel to produce one larger kernel.

The CUDA code generator will then take the kernel description and the kernel source code and perform three tasks: generating the host code, generating the kernel code and finally calculating the CUDA execution configuration. Each of these tasks is now examined in further detail.

Throughout this section several variables will be used within code listings to show items of code that the code generator will replace with suitable values. The following is a list of variables used:

- \$N\$ - Kernel Number.
- \$T\$ - Data Type.
- \$V\$ - Variable Name.
- \$\$ - Variable size (i.e. number of elements in the array).
- \$S1...N\$ - Size of array dimension 1....N.
- \$I\$ - Number of iterations of the loop.

4.3.1 Generation of Host Code

A typical CUDA host program consists of several sections and it is the responsibility of the CUDA host code generator to generate code for each of these sections:

1. Initialisation.
2. Memory allocation
3. Loading data onto the device.
4. Calling the device code.
5. Loading data back from the device.
6. Cleanup.

Initialisation

The initialisation section of code is static and at this stage the code generator simply generates code from the template shown in Listing 4.1. This code firstly queries the number of available CUDA devices and displays an error if this is 0. The final line selects the first available device for use by this application.

Listing 4.1: CUDA: Initialisation Code

```
int noDevicesKernel$$ ;
cudaGetDeviceCount(&noDevicesKernel$$) ;
if (noDevicesKernel$$ <1) {
    printf("No_Cuda_Devices_Found\\n\\r") ;
    exit(1) ;
}
cudaSetDevice(0) ;
```

Memory Allocation

The next step is to generate code to allocate memory on the device. In this stage, code is generated for each variable that is listed as being read from, or written to, in the kernel description. In the case where a variable is listed as being written to and read from, only one line is generated. This code is not generated if the variable is a single variable(i.e. not an array), in this case memory does not need to be allocated, as single datums can be passed directly as parameters to the kernel call, which is shown in Section 4.3.1.

The code to allocate memory on the device is shown in Listing 4.2. This code first defines a pointer to the memory on the device and then allocates it using *cudaMalloc*.

Listing 4.2: CUDA: Allocating memory

```
$T$* $V$Kernel$N$Load;  
cudaMalloc((void*)&$V$Kernel$N$Load, sizeof($T$)*$$);
```

Loading data onto the device

Once the code to allocate memory has been generated, the next step is to load the input dataset into the newly allocated memory. This stage only occurs for variables that are read from by the kernel code.

The code generated by this stage differs if the data being considered is a single dimensional array or a multi-dimensional array. The method used to generate code for each of these circumstances is shown below.

Single Dimensional Array: When a single dimension array is being allocated and loaded onto the device, only a single line of code is generated. This will copy the data from the source variable to the memory on the device. An example of this code is shown in Listing 4.3.

Listing 4.3: CUDA: Loading a single dimensional array onto the device

```
$T$ * $V$Kernel$N$Load;  
cudaMemcpy($V$Kernel$N$Load, $V$, sizeof($T$)*$$,  
          cudaMemcpyHostToDevice);
```

Multi-Dimensional Array: Allocating a multi-dimensional array is effectively a generalisation of allocating a single dimensional array. In CUDA, like in C, multidimensional array are structured as a set of single dimensional arrays linked by pointers. This is demonstrated for a 10 x 10 array in Figure 4.2. When the code generator generates the code for multi-dimensional arrays, it is able to detect

if all dimensions of the array are used. This means that if an array $a[x][y]$ has been declared in the source file, but only one dimension (i.e. $a[10]$) is used in the kernel then only that dimension will be loaded, enabling this array to be treated as a one dimensional array. However, when multi dimensional arrays need to be loaded onto the GPU the code shown in Listing 4.4 is used. In this example, the code first allocates memory space for the pointers to each single dimensional array. Then, the memory for each single dimensional array is allocated, and its data copied. Finally, the array of pointers to the single dimensional arrays is loaded onto the device.

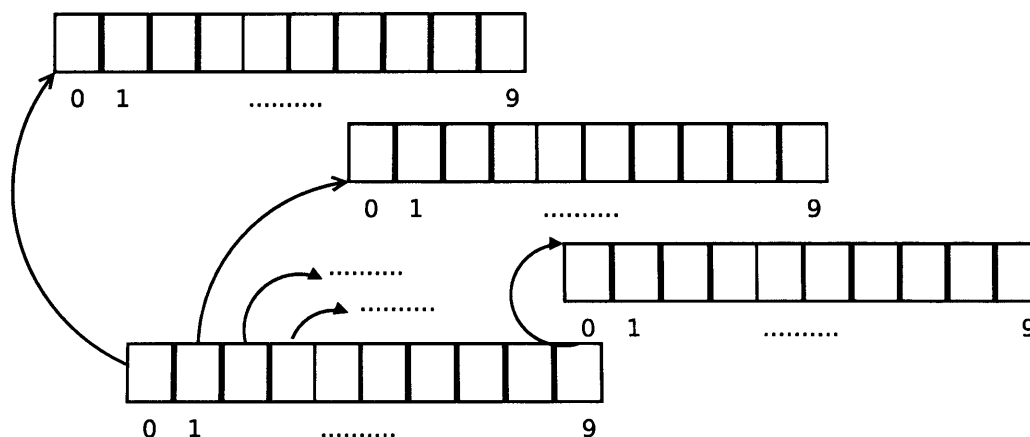


Figure 4.2: A Multi-Dimensional Array

Calling the device code

Once the code to allocate memory and load the data on the device has been generated, the next step is to generate the kernel call itself, the code for this is shown in Listing 4.5. In this code the $\$Parameters\$$ variable represents the list of all single variables that must be passed to the kernel call, pointers to all memory that has been allocated by *cudaMalloc* and a variable containing the number of iterations of the kernel. Additionally, there are two key variables that must be

Listing 4.4: CUDA: Loading a multi-dimensional array

```

$T$ ** $V$Kernel$N$Load ;
cudaMalloc ( (void**)&$V$Kernel$N$Load , sizeof($T$*)*$S1$ ) ;
$T$ ** $V$Kernel$N$LoadTmpDim=($T$*) malloc($S1$ * sizeof
($T$*) ) ;

for ( int loopCountDim1=0; loopCountDim1 < $S1$ ) {
    cudaMalloc ( (void**)&$V$Kernel$N$LoadTmpDim[
        loopCountDim1 ] , sizeof($T$*)*$S2$ ) ;
    cudaMemcpy ( $V$Kernel$N$LoadTmpDim[ loopCountDim1 ] ,
        $V$[ loopCountDim1 ] , sizeof($T$*)*$S2$ ,
        cudaMemcpyHostToDevice ) ;
}
cudaMemcpy ( $V$Kernel$N$Load , $V$Kernel$N$LoadTmpDim , sizeof
($T$*)*$S1$ , cudaMemcpyHostToDevice ) ;

```

generated at this point: D_g and D_b . These variables define how the application is divided between the multiprocessors on the GPU and a description of how these two variables are generated is shown in Section 4.3.3.

Listing 4.5: CUDA: Calling the device code

```

kernel$N$<<<<Dg,Db>>> ( $Parameters$ ) ;

```

One of the key parameters needed to calculate D_g is the number of iterations, this can, in many cases, be calculated from the definition of the loop that formed the kernel, as shown below:

$$\text{Number of Iterations} = \frac{\text{Maximum Bound of Loop} - \text{Minimum Bound of Loop}}{\text{Loop Step Value}}$$

However, it is anticipated that calculating the number of iterations in this manner will not always be possible (i.e. when the loop counter does not increment uniformly), in these cases additional code must be generated prior to the kernel call to calculate the number of iterations at runtime. The generated code will run a dry version (without any loop body) of the loop that simply counts the number of iterations and stores it in a variable. This can then be used to calculate D_g at runtime (D_b is always calculated at compile time). In cases when the loop counter does not change uniformly, code will also need to be generated to provide a mapping from the iteration number to the value of the loop control variable. Once generated, the array containing this mapping must then be loaded onto the device. This code performing this mapping is shown in Listing 4.6.

In cases where both pieces of additional code described above need to be generated, they are merged into one loop.

Listing 4.6: CUDA:Loop counter look-up code.

```

int *xKernel$N$ControlHost=(int*) malloc(sizeof(int)*$I$);

int tmpCounter=0;
$Loop definition that formed natural loop$ {
    xKernel$N$ControlHost[tmpCounter]=$Current Loop
    count value$;
}

```

Loading data back from the device

The next code that needs to be generated will load data back from the device. In this section, the code generator will generate code for each variable in the list of variables being written to. Once again the way this is handled for single dimensional arrays and multi dimensional arrays differ.

For every single dimensional array that is to be copied back, the code shown in Listing 4.7 is generated to copy the data from the device memory to the host memory.

Listing 4.7: CUDA: Loading data back to host for Single Dimensional Arrays

```
cudaMemcpy($V$, $V$Kernel$N$Return, sizeof($T$)*$S$,
  cudaMemcpyDeviceToHost);
```

For multi-dimensional arrays, the code shown in Listing 4.8 is generated, although for the sake of brevity an example using a two dimensional array is shown. This code firstly loads back the memory addresses of each sub-array within the multi-dimensional array. It then, using these memory addresses, loads back each sub array reconstructing them into a two dimensional array on the host.

Listing 4.8: CUDA: Loading data back to host for a two dimensional array

```
$T$ ** $V$Kernel$N$ReturnTmpDim=( $T$* ) malloc( $S1$ *
  sizeof($T$* ) );
cudaMemcpy( $V$Kernel$N$ReturnTmpDim, $V$Kernel$N$Return,
  sizeof($T$* ) * $S1$, cudaMemcpyDeviceToHost );

for ( int loopCountDim1=0; loopCountDim1 < $S1$ ) {
    cudaMemcpy( $V$[ loopCountDim1 ],
      $V$Kernel$N$ReturnTmpDim[ loopCountDim1 ], sizeof
      ($T$)*$S2$, cudaMemcpyDeviceToHost );
}
```

Cleanup

The final section of code that must be generated is to perform cleanup tasks. While the code generator is generating the code in previous sections it will keep a list of all variables that have been allocated with *cudaMalloc* and *malloc*. It

then uses this list to generate *cudaFree* and *free* instructions to free the allocated memory.

4.3.2 Generation of Kernel Code

Once the host code has been generated, the kernel code generator will generate the code for the CUDA kernel from the input kernel code. It does this in two phases; firstly a series of transformations are performed on the kernel code and, secondly, the now transformed code is inserted into a kernel template. Each of these phases will now be examined in more detail.

Code Transformations

The code generator will initially perform the following code transformations on the kernel code, before it is inserted into the kernel template.

- If there are multi-dimensional arrays, where some dimensions have not been loaded (as discussed in 4.3.1), then the unneeded dimensions are removed from the kernel code.
- Any user defined functions must have `__device__` prepended to their definition, and their names changed to differentiate them from their host equivalents.

These code transformations are minimal as CUDA's device API so closely matches C's. However, there is one major area in which they differ: CUDA has no random number generation functionality. This is because using a standard random number generation function would lead to the same sequence of results

being generated in each thread on the device - almost certainly not what the programmer desires.

It was decided, after consideration, that because random number generation is a part of the core API, code should be generated to deal with this case. It was also decided that, at this time, the simplest possible random number generator should be implemented: A Linear Congruence operating with different seeds in each thread (Shown in Listing 4.9). This decision was taken because, although there are better parallel random number generators available such as the Mersenne Twister [86] and Multiply-with-carry [58], the implementation at this stage should be kept as simple as possible and it should be noted that if a user requires higher quality random number generation they would not be using the standard C library random functions in the first place. Additionally, Linear Congruence random number generators are also the type primarily used by ClearSpeed's API [74].

The Linear Congruence random number generator that was selected for use is the one provided as reference in the glibc manual pages [3]. The process of transforming the original random function call takes the following steps:

- The method body for the random number generator must be inserted into the kernel file.
- At the start of the kernel a seed must be declared and initialised.
- Calls to the random function must be replaced with calls to our new generated function, and the seed must be passed as a pointer.

Listing 4.9: CUDA: Random Number Generation

```

//method call to generate random number
__device__ int KernelRand(int *seed) {
*seed = (*seed * 1103515245 + 12345) & 0x7fffffff;
return *seed;

}

//initialise seed
int seed=threadNo * 27 + 13;

// call
int a;
a=KernelRand(&seed);

```

Kernel Template

Once the code transformations have been completed, the kernel code is inserted into the kernel template shown in Listing 4.10. This template shows the version that will be generated when a control array is used to provide a mapping from the iteration number to the loop count value. If the control array is not used the loop count variable value can be calculated as:

$$\text{LoopCountValue} = (\text{LoopStepValue} * \text{IterationNumber}) + \text{MinimumBoundofLoop}$$

In the template the following variables will be used in addition to those described in Section 4.3:

- **\$Kernel Code\$** - The Kernel itself.
- **\$Loop Count Variable Name\$** - The name of the loop counter variable. i.e. i.
- **\$Parameters\$** - The list of parameters passed to the kernel by the host code.

- $D_g.y$ - The y dimension of the grid of thread blocks (See section 4.3.3).
- D_b - The dimension and size of each thread block (See section 4.3.3).

Listing 4.10: CUDA: The Kernel Template

```

__global__ void kernel$N$($Parameters$) {
    int execNo= (((blockIdx.x*$Dg.y$)+blockIdx.y)*
                $Db$)+threadIdx.x;
    int $Loop Count Variable Name$=xControl[execNo];
    if (execNo < NumIterations ) {

        $ Kernel Code $

    }
}

```

4.3.3 Calculating the execution configuration

The CUDA execution configuration consists of four parameters that must be passed to each kernel call [46]:

- D_g , which is the dimension and size of the grid of thread blocks.
- D_b , which is the dimension and size of each thread block.
- N_s , the number of bytes of shared memory that is dynamically allocated per block.
- S , any associated CUDA streams.

As the current version of the CUDA code generator does not utilise dynamically allocated shared memory or CUDA streams, the final two parameters can be

ignored and be allowed to take their default values of zero. However, the first two parameters need to be calculated. Before this can be done it is important to understand how CUDA allocates each execution of the kernel (known as a thread) across the graphics card. The two key virtual groupings of threads [46] that CUDA utilises are defined as::

Thread Block: A thread block is a group of threads that can cooperate together. Thread blocks are processed in batches and each thread block is executed by a single multiprocessor. The number of blocks that a multiprocessor can process is dependent on the register and shared memory usage of the kernel being executed.

Warp: A group of threads from a thread block that is executed by a multiprocessor in a SIMD fashion. The current active warps i.e. all the warps from the currently active thread blocks on the multiprocessor are time-sliced to make maximum use of the multiprocessor computational resources[108].

From these definitions there are some general rules for efficient CUDA execution that should be followed:

- Each multiprocessor should have enough threads available to it to ensure it is fully occupied.
- There should be at least as many thread blocks as there are multiprocessors. Ideally twice as many.
- The number of threads per block should be a multiple of the warp size.

From these rules it can be seen that D_b must first be calculated to determine how many threads per block are required to ensure that each multiprocessor is fully occupied. Once D_b is known then D_g can be calculated using D_b .

Calculating D_b

To assist programmers in developing applications that maximise utilisation, NVIDIA have provided an Excel spreadsheet [104] to calculate device occupancy. Formulas from this spreadsheet are used to calculate D_b by the porting system. It should be noted that, in my implementation, the hardware dependant values used in these formulas are stored in a configuration file, this enables the CUDA back-end to easily be reconfigured to cope with differing GPU models.

The utilisation of each multiprocessor is determined by the number of active thread blocks on the multiprocessor and the number of warps per block, such that:

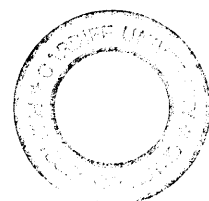
$$Occupancy = \frac{NumActiveThreadBlocks * NumWarpsPerBlock}{WpMP} \quad (4.1)$$

Where **WpMP** represents the hardware limit for the number Warps Per Multi Processor.

The value of D_b depends primarily on the value of *NumWarpsPerBlock*. So the optimum value for this must be first be calculated.

To calculate *NumWarpsPerBlock* we must first calculate a value for *NumActiveThreadBlocks*. However, due to the nature of the equations involved it is not possible to directly compute the value of *NumActiveThreadBlocks* for a specific application. However, the optimum value of *NumActiveThreadBlocks* for the specific GPU that is being used can be calculated, by assuming that:

$$NumActiveThreadBlocks = BpMp$$



Where $BpMp$ is the maximum number of active blocks allowed by the hardware. Making this assumption allows the calculation of the occupancy as:

$$Occupancy = \frac{BpMp * NumWarpsPerBlock}{WpMP} \quad (4.2)$$

As we require maximum occupancy(meaning that $Occupancy = 1$) we now have the following equation:

$$NumWarpsPerBlock = \frac{WpMP}{BpMP} \quad (4.3)$$

This value is the minimum number of warps required to achieve full occupancy of each multiprocessor for this GPU regardless of application. However, due to features of the application being executed, such as shared memory and the number of registers used, $NumActiveThreadBlocks$ may take a value lower than that of $BpMp$ and possibly this value may be as low as 1. This means that the actual number of warps takes a value:

$$\frac{WpMP}{BpMP} \leq NumWarpsPerBlock < WpMP \quad (4.4)$$

The only way to accurately determine the optimum value for $NumWarpsPerBlock$ and $NumActiveThreadBlocks$ for an application is, taking each value of $NumWarpsPerBlock$ that is within the range shown above, compute $NumActiveThreadBlock$ manually and then recompute the occupancy using Equation 4.1.

To compute $NumActiveThreadBlocks$ manually the minimum value from Equations 4.6 and 4.5 are taken.

$$\lfloor \frac{WpMP}{NumWarpsPerBlock} \rfloor \quad (4.5)$$

Where **WpMP** is the hardware limit for the number of Warps per multiprocessor.

$$\lfloor \frac{ITr}{64 * NumRegistersUsed * (NumWarpsPerBlock - \lfloor \frac{NumWarpsPerBlock}{2} \rfloor)} \rfloor \quad (4.6)$$

Where **ITr** is the hardware limit for the total number of registers per multiprocessor and *NumRegistersUsed* is discovered by examining the kernels CUBIN file. This CUBIN file is generated by the compiler and an example is shown in Figure 4.3.

```
architecture {sm_10}
abiversion {0}
code {
    name = _Z5nbodyPfs_S_S_S_i
    lmem = 0
    smem = 60
    reg = 17
    bar = 0
}
```

Figure 4.3: A CUBIN file

Once all the values of *NumWarpsPerBlock* have been computed, the desired value will be the one that has the highest occupancy. In the case when there are several values with the same occupancy the lowest value of *NumWarpsPerBlock* shall be taken so that the maximum number of thread blocks can be formed from the problem size.

Once the value for *NumWarpsPerBlock* has been computed we can use the following equation from [104]:

$$NumWarpsPerBlock = \lceil \frac{D_b}{TpW} \rceil \quad (4.7)$$

Where **TpW** is the hardware limit for the number of threads per warp.

As we require the smallest amount of threads to form the number of warps desired, while still ensuring that D_b is a multiple of the warp size. D_b can be written as:

$$D_b = (WarpsPerBlock) * TpW \quad (4.8)$$

Calculating D_g

With D_b calculated, calculating D_g is relatively simple. As the CUDA porting system has already determined the number of iterations of the loop that is forming the kernel. (Let this be called *NumIterations*). Then we can say the number of thread blocks should be:

$$D_g = \frac{NumIterations}{D_b} \quad (4.9)$$

However, as D_g in theory could be a very large number and CUDA sets a hard limit for the size of each dimension of the grid. We need to split D_g into X and Y coordinates for the grid. This is done as follows:

$$D_{g.x} = \lceil \sqrt{\lceil \frac{NumIterations}{D_b} \rceil} \rceil \quad (4.10)$$

$$D_{g.y} = \lceil \frac{NumIterations}{D_b * D_{g.x}} \rceil \quad (4.11)$$

It should be noted that this will often mean that more iterations than required are run. This is a side effect of splitting the kernel uniformly across the GPU

and it is preferable to executing fewer iterations than required. In order to deal with the problems that executing additional iterations of a kernel may cause, a branch is present in the CUDA kernel template (Listing 4.10) to ensure that these additional kernels perform no computation.

In this section we have covered the entire process undertaken to generate device specific code for NVIDIA's C for CUDA language. A worked example of the process that has been described here is shown in Section 4.6.

4.4 Porting to C_N

The second back-end that has been developed for the system is C_N for ClearSpeed. C_N is a modification of the C language with two added keywords:

Mono: The mono keyword designates that a variable is to be stored in the devices main memory (A non parallel variable).

Poly: The poly keyword designates that a variable is to be stored in the memory of the particular processing element on which the current instantiation of the kernel is executing on (A parallel variable).

Due to the differences in the ClearSpeed's architecture, the ClearSpeed code generator is significantly different to CUDA's. ClearSpeed has no context switching mechanism allowing memory IO latency to be hidden, so the problem of memory latency must be dealt with explicitly by the programmer, using asynchronous I/O and other methods. This adds considerably to the complexity of the code generator.

The recommended method to increase memory IO efficiency on ClearSpeed

and thus reduce the IO latency is to reduce the number of data transfers and increase the size of each transfer [98]. The simplest way to achieve this is to use double buffering, which enables the system to leverage ClearSpeed's ability to perform asynchronous IO. The decision to use double buffering had led to the ClearSpeed code generator having four modes of operation: Generating Host Code, Generating a Kernel Template, Generating Code for a Non-Buffered Kernel and Generating code for a Buffered Kernel.

In order to select which kernels are buffered or not, the following two rules are applied, depending on the number of subkernels that the kernel being executed has:

1. If the kernel being executed has no sub-kernels then it will be a buffered kernel.
2. If the kernel has subkernels then the innermost kernel(i.e. at the bottom of the kernel tree), will be a buffered kernel, while the remaining kernels will be rolled up into one non-buffered kernel.

However, it should be noted that not all kernels are able to make use of the buffering technique and any kernels with large poly memory requirements or where either the memory read/writes are not incremental between kernel executions can not be buffered.

Each of following three modes of operation are outlined in the following sections.

4.4.1 Generation of Host Code

The ClearSpeed host code generator operates in largely the same manner as the CUDA code generator, however there are some important differences. The

code generated by the ClearSpeed host code generator can be classified into the following four sections:

1. Initialisation.
2. Loading data onto the device and calling the device code.
3. Loading data back from the device.
4. Cleanup.

Also, throughout this section several variables will be used within code listings to show items that the code generator will replace with suitable values. The following is a list of variables used:

- $\$N\$$ - Kernel Number.
- $\$T\$$ - Data Type.
- $\$V\$$ - Variable Name.
- $\$S\$$ - Variable size(i.e. number of elements in the array).

Initialisation

The main difference between the initialisation within the ClearSpeed code generator, apart from syntax differences, occurs because each ClearSpeed accelerator consists of multiple ClearSpeed chips. The initialisation section must, once the API has *connected* to the card, discover the number of chips present on the card, this is shown in Listing 4.11.

Once this has been done, the number of times the kernel will be executed must be computed, if necessary, and the control array that contains the mapping from iteration number to the value of the loop control value must be populated. These sections of code are identical to that shown for CUDA in section 4.3.1.

Finally, using the previously computed values, the number of executions per chip and per processing element must be calculated:

$$IterationsPerProcessor = \lceil \frac{NoIterations}{NoChips} \rceil$$

$$IterationsPerProcessingElement = \lceil \frac{IterationsPerProcessor}{NumProcessingElements} \rceil$$

Listing 4.11: ClearSpeed: Initialisation Code

```

int noDevicesKernel$$ ;
struct CSAPIState* kernel$$State=NULL;
CSAPI_num_cards(&noDevicesKernel$$);
if (noDevicesKernel$$ < 1) {
    printf("No_Clearspeed_Devices_Found\n\r");
    exit(1);
}
kernel$$State=CSAPI_new();
CSAPI_connect(kernel$$State, CSH_Private, CSC_Direct, "
    localhost", CSAPLINSTANCE_ANY, 0);
int noProcessorsKernel$$;
int noPeKernel$$;
CSAPI_num_processors(kernel$$State, &
    noProcessorsKernel$$);
CSAPI_num_pes(kernel4State, 0, &noPeKernel$$);

```

Loading data onto device and calling the device code

Due to the presence of multiple chips on each ClearSpeed accelerator board, the memory allocation, loading of data and calling the kernel must be performed on a per chip basis.

Firstly a set of variables need to be declared to store process handles for the running kernel on each chip, this is shown in Listing 4.12.

Listing 4.12: ClearSpeed: Declaring Process Handles

```
struct CSAPIProcess *process$N$ [noProcessorsKernel$N$];
int procNo$N$;
```

The next step is to declare variables to hold pointers to all memory that will later be read back from the device. Additionally, the maximum and minimum values within that memory that are accessed by each chip must be stored. Finally, a single pointer must be declared to each array that will be loaded onto the device. All this needs to be done outside of per-chip loop so that this data can be accessed later in the program. The code generated for each variable written to by the device is shown in Listing 4.13.

Listing 4.13: ClearSpeed: Declaring Variable Pointers

```
CSAPIMemoryAddress $V$ReturnKernel$N$ [
    noProcessorsKernel$N$];
int writeMax$V$Kernel$N$ [noProcessorsKernel$N$];
int writeMin$V$Kernel$N$ [noProcessorsKernel$N$];
CSAPIMemoryAddress symbolAddr$V$Kernel$N$;
```

The first task that must be performed is loading the device code onto the chip. This must be done on a per chip basis, but all the compiled device code must be

loaded onto all chips before any further memory allocation can occur. The code for initialising a chip is shown in Listing 4.14.

Listing 4.14: ClearSpeed: Loading the Device Code

```
CSAPI_load(kernel$$State,procNo$$,"kernels/kernel$$.  
csx",NULL,&(process$$[procNo$$]),CSAPI_NO_TIMEOUT);
```

At this point it should be noted that, unlike CUDA, ClearSpeed's requirement that a CSX program must first be loaded onto a chip before any memory can be allocated, precludes the ability for ClearSpeed accelerators to keep data on the chip between executions of different kernels.

The remainder of the set up code is contained within a single loop and is done on a per chip basis. Once the code to load the device program has been executed, a second loop is created to allocate memory and load data into the device. This process consists of two steps: first acquiring the memory address of the desired variable on the ClearSpeed chip, and then copying data to it. If the variable is a single variable (i.e. not an array) then there is no need to allocate memory. If the variable is an array, then the memory must first be allocated. An example of fetching the memory address and then copying data to it is shown in Listing 4.15 and an example of allocating memory and then copying data to it is shown in Listing 4.16.

Multi-dimensional arrays in ClearSpeed are handled in identical fashion to that of CUDA (shown in Section 4.3.1) using the *CSAPI_allocate_shared_memory* and *CSAPI_write_mono_memory* methods.

Listing 4.15: ClearSpeed: Loading Single Variables

```

CSAPIMemoryAddress symbolAddr$V$Kernel$N$ ;
//fetch memory address
CSAPI_get_symbol_value ( kernel$N$State , process$N$ [
    procNo$N$ ] , "$V$" , &symbolAddr$V$Kernel$N$ ) ;
//copy data
CSAPI_write_mono_memory ( kernel$N$State ,
    CSAPLTRANSFER_PARAMS_SAFE , symbolAddr$V$Kernel$N$ ,
    sizeof ($T$) , &$V$ ) ;

```

Listing 4.16: ClearSpeed: Loading Arrays

```

CSAPIMemoryAddress symbolAddr$V$Kernel$N$ ;
//allocate memory
CSAPI_allocate_shared_memory ( kernel$N$State , procNo$N$ ,
    CSM_Dram , sizeof ($T$) * $$$ , sizeof ($T$) , process$N$ [
    procNo$N$ ] , "$V$" , &symbolAddr$V$Kernel$N$ ) ;
//copy data
CSAPI_write_mono_memory ( kernel$N$State ,
    CSAPLTRANSFER_PARAMS_SAFE , symbolAddr$V$Kernel$N$ ,
    sizeof ($T$) * $$$ , $V$ ) ;

```

When considering memory allocation on a ClearSpeed device there are however additional concerns. Each ClearSpeed chip consists of a DRAM memory store of size X MBytes. This memory however is split between each chip on the ClearSpeed card and so each chip is only able to allocate $\frac{X}{\text{NumberOfChips}}$ MBytes of the total memory.

Even though the main memory is split between chips; a chip is still able to access memory attached to another chip, but a performance penalty is incurred. From experience, it has been determined that when possible it is preferable to duplicate the input data-set into each chips memory rather than incur this performance penalty.

All of these factors must be taken into account when generating code to allocate memory on a ClearSpeed device. Whenever possible, the ClearSpeed code generator will duplicate the required data, into each chip's own segment of memory. This, however, dramatically reduces the total memory available to the application and there are certain circumstances where this strategy is not possible.

To combat this, the code for allocating memory is generated in the pattern shown in Listing 4.17, in this listing detailed implementation details are omitted for brevity, but can be seen in the worked example shown in Section 4.6.

Listing 4.17 illustrates the memory allocation operating in two modes:

1. Each array is allocated onto one chip, and pointers are loaded onto the other chips: This enables the application to make full use of the ClearSpeed card's memory, but incurs a performance penalty.
2. Each array is duplicated onto each chips own segment of memory. This means

that application only has $\frac{X}{\text{NumberOfChips}}$ of the total memory available to it but gives improved performance.

When reading the code it should be noted that it is designed in such a way to enable the first chip to start executing as soon as the first iteration of the per chip loop has run, rather than having to wait for all memory to be allocated across all chips before starting execution.

The above memory allocation strategy only applies for data that is read and not written to. In the case of data that is written to; the output data-set will either be split based on the iterations that are taking place on each chip, or the output data-set will be loaded solely onto one chip, and then a pointer to it will be loaded onto all other chips.

Once the memory has been allocated and copied, then the execution of the kernel must be started. Before this can be done two variables holding the number of iterations that the current chip is performing and the number of the first iteration executing on the chip must be loaded into the chip's memory. Once this is done the kernel is launched using the code shown in Listing 4.18.

Loading data back from the device and cleanup

The code for loading back data from the device to the host, is once again structured in a loop acting on a per chip basis. Firstly the loop must wait for the chip being considered to have finished executing, this is shown in Listing 4.19. Then, once the chip has finished executing, data can be loaded back into the host memory, obviously considering that if only a portion of the array was loaded onto the device then the portion loaded back will need to be positioned correctly within

Listing 4.17: ClearSpeed: Allocating Memory

```

//A flag to hold what memory allocation mode we are in
int doubleload=0;
for (i=0; i< $Num Processors$; i++) {
//Variable to store total number of bytes allocated
int totalAllocated=0;

//Allocate array A of size X bytes
totalAllocated+=X;
//Determine what chip A should be loaded on
int procA=floor(totalAllocated/$Mem Per Chip$);
if (doubleload==0) {
    //Not Double Loading
    if (i == 0) {
        //Memory loading is all done on the first
        //pass of the loop
        //So load array A into procA's memory
    }
    if (procA!=i) {
        //If we haven't already loaded the data
        //into this processors memory
        //Load a pointer to the data into this
        //processors memory
    }
} else if (doubleload==1) {
    //Double Loading
    //So load the data set into chip i
}

//At the end of the iteration of the loop we check if we
//have used all the chips memory.
//If we haven't then we can duplicate the data-set across
//all chips.
//This only has any effect in the first iteration.
if (totalAllocated < $ Mem Per Chip$) doubleload=1;
}

```

Listing 4.18: ClearSpeed: Calling a Kernel

```
CSAPI_run( kernel$N$State , process$N$ [ procNo$N$ ] , NULL );
```

the main dataset on the host. This is done using the *CSAPI_read_mono_memory* function demonstrated in Listing 4.20.

Listing 4.19: ClearSpeed: Wait for execution to finish

```
CSAPI_wait_on_terminate( kernel$N$State , process$N$ [
  procNo$N$ ] , CSAPLNO_TIMEOUT );
```

Listing 4.20: ClearSpeed: Loading Data Back onto the Host

```
CSAPI_read_mono_memory( kernel$N$State ,
  CSAPLTRANSFER_PARAMS_SAFE, $V$ReturnKernel$N$ [
  procNo$N$ ] , $$*sizeof($T$) , $V$ );
```

The final code generated performs any required cleanup. The only two tasks that need to be performed here are calls to *free* to deallocate any local memory that has been allocated, and a single call to free the ClearSpeed card (which also frees the card's shared memory), this call is shown in Listing 4.21.

4.4.2 Generation of Device Kernel Template

The first stage in generating C_N device code is to generate the kernel template. The kernel template is only generated for the top level kernel of the set of kernels that are being ported.

This template consists of the generation of variable declarations for all data loaded onto the device as *mono* (this *mono* keyword is omitted as ClearSpeed assumes

Listing 4.21: ClearSpeed: Cleanup

```
CSAPI_delete(kernel$N$State);
```

mono unless poly is specified) variables which is shown in Listing 4.22, and the generation of a main method which is shown in Listing 4.23. Once the kernel template has been generated, the code for the actual kernel itself is inserted using *#include* directives. Equivalently any sub kernels are also included by generating *#include* directives at appropriate points within the kernel code itself.

Listing 4.22: ClearSpeed: Declaring Global Variables

```
//a single variable
int noPerProc;
// an array
float * data
```

Listing 4.23: ClearSpeed: Kernel Template

```
int main(int argc, char**argv){
//an example include directive to include the actual
kernel code.
#include "kernels/kernel$N$.cn"
}
```

4.4.3 Non-Buffered Kernels

Once the kernel template has been generated, the code for the kernel itself must be generated. This consists of a series of additions and transformations on the input code.

Firstly, as the code generated will execute exactly X times, where X is calculated as:

$$X = \text{NumberOfChips} * \text{NumberOfProcessingElementsOnChip}$$

We must generate code that allows us to deal with situations where the number of iterations are less than the number of times the generated code will execute. This code is shown in Listing 4.24. In this code the *if* statement within the for loop is present to deal with situations where the number of iterations does not divide evenly onto the topology of the device.

Listing 4.24: ClearSpeed: Multiple Iterations per PE

```

int kernel$N$Loop;
poly int offset;
noPerProc=ceil((double)noExec/pePerChip);

offset=(get_penum()*noPerProc)+firstExec;

for (kernel$N$Loop=0;kernel$N$Loop < noPerProc;
      kernel$N$Loop++) {
if (offset+kernel$N$Loop < noExec) {

}
}

```

The remainder of the kernel code will be inserted inside the loop and if statement, before this can be done however it must undergo a series of transformations:

Variable Declarations: Each variable declared inside the kernel code must be converted into a poly variable declaration. To do this the code generator will add the keyword *poly* before each such variable declaration.

Functions: Each function call within the kernel code, that is not a function defined in the program code itself, needs to be converted to a poly function call. To do this the code generator will change the name of the function adding a p to the end. i.e. *sqrtf* will transform to *sqrtfp*. There are, however, some more complex transformations that need to occur here.

One of these cases is that of random number generation. When a call to function that generates random numbers i.e. *rand* or *random* is detected the ClearSpeed parallel random number generation API must be used. Using this API consists of [74]:

- Adding the library flag *-lcn_rng* to the makefile,
- Adding an include directive to the top of the kernel source file,
- Adding a method call to initialise the random number generator,
- Replacing the old function call with a new functional call to the ClearSpeed random number generator.

An example of the code described above is given in Listing 4.25.

Additional Include Files: As mentioned in the previous section, each function call must be translated to its poly equivalent, enabling it to operate on poly variables. This means that the `#include` directives must also be changed to allow the importing of the poly function prototypes. To achieve this a p is added to the end of the file name of the include file and the new directive is added in addition to the existing one. i.e. `#include <stdio.h>` will cause `#include <stdiop.h>` to be generated.

Listing 4.25: ClearSpeed: Random Number Generation

```
//include directive
#include <rngp.h>

//code to initialise random number generator
//Using the ClearSpeed rand48(A Linear Congruence random
number generator).
poly cs_rand48_state rng_state;
cs_rand48_stream rng_stream;

// this random number generator uses a seed of
// 13 + pe number * 27
cs_init_rng_multiseed(rand48,&rng_stream,&rng_state,13+
    get_penum()*27);

//code to generate a random number

poly long a;
a=(cs_frand48(&rng_stream)*RAND.MAX);
```

Reading Mono Memory: As each ClearSpeed processing element is only able to access data within its own poly memory, every time the program code requires data from mono memory, code must be generated to copy the required datum from mono memory to poly memory. When generating this code, the code generator ensures that each datum is only loaded from mono memory the first time it is accessed, after that the copy in poly memory is always used. This process consists of several steps, all of which are illustrated in Listing 4.26:

1. A temporary variable and a semaphore must be declared at the top of the block.
2. A call to *async_memcpy2p* must be generated, before the variable is used and as early as possible within the code. This is to give the maximum amount of time for the data transfer.
3. Immediately before the datum is needed, the program must wait to ensure the semaphore is ready.
4. The variable name within the code must be altered to the temporary variable.

Listing 4.26: ClearSpeed: Reading from mono memory

```

//declare semaphore
mono short SEMAPHORE=1;
//declare temporary variable
poly float tmpKernel0;
//generate function call
async_memcpy2p(SEMAPHORE,&tmpKernel0,$data array$+(
    $position in array$ ),sizeof($type$));
//wait for semaphore
sem_wait(SEMAPHORE)

```

Writing to Mono Memory: Writing to mono memory is similar to reading from mono memory. A temporary variable and semaphore must still be declared

and the variable name within the code must be changed to match the temporary variable. However, the position of the generated function call and the function call itself are different. The function call is shown in Listing 4.27 and in this case it is generated immediately after the final time that variable is used in the current iteration of the kernel. Finally, the *sem_wait* is inserted immediately before the first use of the variable in the next iteration of the kernel. This gives the memory transfer the maximum amount of time to complete without it blocking the program. However, this method does present a slight problem, as in certain cases a *sem_wait* will be encountered before any call to *async_memcpy2m*, this would mean the program would block infinitely, as without a memory transfer in progress the semaphore will never become ready. This is solved by generating a *sem_sig*, which initially sets the semaphore's state to ready.

Listing 4.27: ClearSpeed: Writing to mono memory

```
async_memcpy2m(SEMAPHORE, $data array$+($position in
  array$ ),&tmpKernel0, sizeof($type$));
```

There is one exception to these final two transformations, if the variable that is being read from or written to, is a single variable (not an array) then they are not necessary. ClearSpeed implicitly allows each PE to access single variables stored in mono memory [35].

4.4.4 Buffered Kernels

Generating code for a buffered kernel, consists of several main sections. A template for a buffered kernel is used, this consists of calls to several functions that will also be generated by the system. During this section we refer to two variables, *BufferSize*, which is the size of each buffer and *NumBuffers*, which is the number of buffers. These variables are discussed further in Section 4.4.5.

The overall buffer template is described in Listing 4.28, within the listing the following assumptions are made, in order to simplify the code for presentation purposes:

- All input data are of the same type.
- The input and output semaphore are declared as ISEMAPHORE[.] and OSEMAPHORE[.] respectively.
- The input and output buffers are declared as inputBuffer[0..1] and outputBuffer[0..1] respectively.
- The Semaphores on the output buffers are set in the ready state.

The following variables are used within the listing:

- X - The number of individual memory copies that must be done to populate a buffer.
- Y - The number of individual memory writes that must be done to empty an output buffer.
- T - Data type of input.
- N - Kernel Number.

A buffered-kernel consists of calls to several functions, which are generated specifically for the kernel:

saveData and loadData: These functions start the loading of data from either mono to poly or poly to mono memory. This data, while in poly memory, are

Listing 4.28: ClearSpced: Buffered Kernel Template

```

mono int currentBuffer=0;
mono int bufferIter=0, bufferOffset=0;

//start loading the first two buffers
loadData$$$(ISEMAPHORE[ currentBuffer ], inputBuffer [
    currentBuffer ], 0);
currentBuffer = 1;
loadData$$$(ISEMAPHORE[ currentBuffer ], inputBuffer [
    currentBuffer ], 1);

for ( bufferIter=0; bufferIter < $NoBuffers$; bufferIter++)
{

int innerLoopIter;
if ( currentBuffer==0) currentBuffer=1; else currentBuffer
    =0;
//block until the buffer we are about to process is
    loaded into memory
inBufferWait$$$(ISEMAPHORE[ currentBuffer ]);

//is the output buffer ready to use.
outBufferWait$$$(OSEMAPHORE[ currentBuffer ]);

for ( innerLoopIter=0; innerLoopIter < $BufferSize$;
    innerLoopIter++) process$$$(inputBuffer [ currentBuffer
    ], outputBuffer [ currentBuffer ], bufferOffset+
    innerLoopIter, $List of all parameters required$,
    innerLoopIter);

//start saving the data back to mono memory
saveData$$$(OSEMAPHORE[ currentBuffer ], outputBuffer [
    currentBuffer ], bufferIter, bufferIter+2);

//once processing is finished start reloading this buffer
    only if more buffers need to be filled
if ( bufferIter +2 < $NoBuffers$) {
    loadData$$$(ISEMAPHORE[ currentBuffer ], inputBuffer
        [ currentBuffer ], bufferIter+2);
}

bufferOffset+=BUFFERSIZE;

}

```

stored in buffers. An example of a buffer for a kernel where each iteration accesses one datum from arrays a,b and c and the buffer size is five is shown in Figure 4.4.

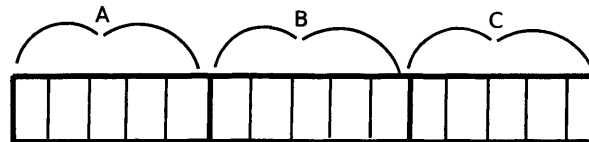


Figure 4.4: ClearSpeed: Memory buffer layout

This method obviously places limitations on when using a buffered kernel is applicable. If kernels do not access memory in an incremental fashion, it will lead to loading of additional data which are not required. This will cause, in some cases, major performance issues as the data communication path becomes saturated with unneeded data transfers.

outBufferWait and inBufferWait: These functions force the program to wait until the buffer has finished saving from poly memory into mono memory or from mono to poly, respectively. To do this, it contains a loop which performs a *sem_wait* for each memory transfer that has been conducted.

process: The process function contains the actual kernel code itself. The code here will have undergone transformations to its variable declarations and function calls as described in Section 4.4.3. Additionally, all read/writes to variables that are now stored in the buffer must be translated to refer to their location within the buffer and not to the device's main memory. As each data within the buffer is arranged in order of first appearance within the code (note, in this case, each unique data access is considered separate, even if it is in the same array. i.e. `data[z]` and `data[y]` will be considered as two data accesses). Then, knowing the location within the code, the location of the data within the buffer can be calculated as:

$$(X * BufferSize) + IterationNumberWithinBuffer$$

Where X is a value representing the order of occurrence within the code.

In certain kernels, additional code must be generated to deal with circumstances where there are variables that must be loaded from mono memory but do not change between iterations of the kernel. These are known internally by the system as *invariants* and mainly occur when the buffered kernel is a subkernel of a non-buffered kernel. When invariants are present, code is generated to load all of these into an invariant array before the first iteration of the kernel is executed. This invariant array is then passed to the process function.

Even though the code described here generates, for organisational purposes, pieces of code in separate functions, it is entirely possible for the code to operate inline with the main code running on the device. This is desirable in certain cases, i.e. where the application struggles to fit in the memory available on the PE.

4.4.5 Determining Buffer sizes

In order to provide the best possible performance, the buffer size and the number of buffers used within a buffered kernel must be calculated.

It is obvious that these two variables are related such that:

$$NumberOfIterationsOnProcessingElement = BufferSize * NumberOfBuffers$$

So in practice only the *BufferSize* will be calculated.

The main objective in using buffering, is that it decreases the total number of memory copy operations, while increasing the size of each individual copy operation. On a ClearSpeed chip maximum transfer rates are only achieved using higher byte per transfer sizes [98]. It is known from previous experiments carried out [98] that the achievable bandwidth begins to approach its maximum for mono to poly transfers when the number of bytes per transfer equals 128. For poly to mono transfers, the figure is 256 bytes.

From this, it can be said that the size of each data transfer is:

$$SizeofDataTransfer = BufferSize * sizeof(DataType)$$

So, a value of *BufferSize* is selected so that each transfer moves enough data to achieve its peak performance. However, there is a limitation to this; each ClearSpeed PE has only 6kbytes of poly memory and we assume that that only 5kbytes are available for us to allow memory space for the call stack. So the total amount of memory available for all input and output for each buffer is (in bytes).

$$MemoryAvailable = \frac{(5120 - SizeofInvariants)}{2}$$

This means that in cases where there is not enough poly memory to have a value of *BufferSize* that achieves maximum efficiency, then *BufferSize* is set to the largest possible value that will fit into memory. In cases where there is sufficient poly memory then the *BufferSize* will be the smallest possible value, where all transfers meet the requirements to achieve maximum bandwidth.

This section has outlined the process that is undertaken to port to ClearSpeed's C_N language. Due to the architectural differences between ClearSpeed and C for CUDA, the generation of device specific code for C_N is a more complex process.

An example of this porting process is shown in Section 4.6 and it is interesting to see the difference in the size of the program code generated by each of the currently available back-ends.

4.5 Creating Build Scripts

The final stage of the compilation process, for both of the two device types considered, is the generation of build scripts. This consists of generating a Makefile, that will allow the automatic building of the generated code using Make.

For CUDA this consists of generating rules to allow the compilation of *cu* files to *c* files, then the compilation and linking of *c* files into the final binary and the linking of this binary to the CUDA runtime library. An example of a CUDA Makefile is shown in Listing 4.29.

Listing 4.29: CUDA: Makefile

```
OBJS=src//src/dgemm.o
CFLAGS=-I.
%.c:%.cu
    nvcc $(CFLAGS) -cuda $< -o $@
%.o:%.c
    cc $(CFLAGS) -c $< -o $@
all: $(OBJS)
    cc -L/usr/local/cuda/lib $(OBJS) -O4 -o ./run-gpu
    -lcudart
clean:
    rm -f run-gpu
    rm -f $(OBJS)
```

For ClearSpeed the Makefile consists of rules to build the *.csx* (Device binary file) from the kernel code and then build and link the C host code, ensuring that it

is linked with the ClearSpeed library and the operating system dynamic linking library (libdl in the case of Linux). An example of a ClearSpeed Makefile is shown in Listing 4.30.

Listing 4.30: ClearSpeed: Makefile

```

OBS=src//src/dgemm.o
CNOBS=kernels//kernel3.csx
CFLAGS=-I. -I/opt/clearspeed/include/host
%.csx:%.cn
    cscn --dynamic -O4 $< -o $@
%.o:%.c
    cc $(CFLAGS) -c $< -o $@
all: $(OBS) $(CNOBS)
    cc -L/opt/clearspeed/lib $(OBS) -o ./run-cs -
        lcsapi -ldl -lm
clean:
    rm -f run-cs
    rm -f $(OBS)
    rm -f $(CNOBS)

```

4.6 Code Generation Example - GEMM

This section will outline a complete example of the porting process for a simple application code: a general matrix multiplication. Firstly describing the initial analysis and parsing that was performed by the client (this process was outlined in Chapter 3) and then describing the process undertaken to port the application to both C for CUDA and C_N .

The input source of the application is shown in Appendix A.1. This source, once passed to the system client, generates the kernel tree that is shown in Figure 4.5.

Figure 4.5 shows that the client detects six possible kernels, each one correspond-

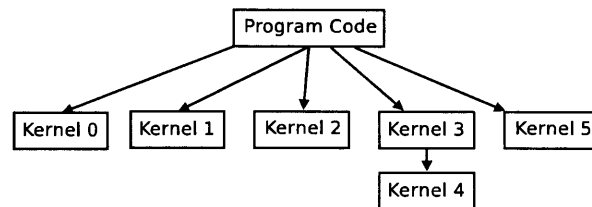


Figure 4.5: GEMM: Example Kernel Tree

ing to a *for* loop within the code. Once parsing has been completed, kernels 0,1,2 and 5 will all be discounted by both the ClearSpeed and CUDA back-ends because they each contain function calls (*fgets* and *printf*) that correspond to I/O, something that neither of our considered devices are able to handle.

At this point the application now consists of the two kernels that are acceleratable and the remainder of the code that will be executed on the host. The next step is to actually port the code. The code that will run on the CPU (shown in Appendix A.2) will be the same for both of the considered devices. However, actions taken by the C_N and CUDA back-ends from this point differ significantly.

CUDA: The first action that the CUDA back-end will take will be to combine kernels 3 and 4 into one larger kernel. This kernel will then be ported to CUDA. The host code is shown in Appendix A.3.1 while the device code is shown in Appendix A.3.2.

C_N : The ClearSpeed back-end will operate differently; firstly using the process outlined in this chapter, the ClearSpeed back end will firstly generate the host code, which is shown in Appendix A.4.1. The key difference to note here is that the C_N host code contains the loops that enable the application to utilise both chips on the ClearSpeed Accelerator.

Secondly, the device code is generated for both kernels. As kernel 3 has a sub-kernel it is ported as a non-buffered kernel and the code is shown in Appendix

A.4.2. Kernel 3 is also the top level kernel, so it contains the *main* method for the device program. Kernel 4, as it is the innermost kernel, is generated as a buffered kernel and the code is shown in Appendix A.4.3.

When run on both of the GPU and ClearSpeed device, the application produces results that match those produced by the CPU and also, depending on data-set size, provides improved performance compared to the CPU.

4.7 Chapter Summary

This chapter has covered the functionality of one of the key requirements for raising the level of abstraction available for programming acceleration devices: the ability to perform source(C) to source(Accelerator) translation from a standard language to a language/API suitable for execution on an acceleration device.

This chapter has covered the two back-ends that have currently been developed, CUDA-C and C_N for ClearSpeed. For each of these back-ends there is functionality in place to:

- Generate code to manage the loading of data to and from the device.
- Generate a kernel code to execute on the device.
- Generate appropriate build scripts to allow the automated compilation of the generated code.

While it can be seen from the work presented in this chapter that there are many similarities between C_N and CUDA, it can be also been seen that porting to C_N is a much more involved process than porting to CUDA. This is largely due to the

abstraction at which the memory model of the ClearSpeed chip is exposed to the developer. Using C_N the user has to explicitly manage data movement between mono and poly memory using memory copy functions, in comparison CUDA hides explicit data movement within the memory hierarchy from the programmer.

Possibly the most important difference from a performance perspective is CUDA's ability to natively hide memory IO latency from the programmer by time slicing between a large number of threads. This ability is not present on a ClearSpeed device and C_N programmers must use their own methods to hide the memory latency within their programs. This performance issue is compounded by ClearSpeed's accelerators having their main memory split between chips. This further complicates programming, forcing the developer to either split or duplicate data between the two chips or incur the steep performance penalty of having a chip accessing memory attached to one of the other chips on the device.

Other differences, include CUDA's ability to keep a data-set stored in GPU memory between execution of kernels, this is especially useful in applications with pipeline characteristics. Although ClearSpeed does not support this, it can be achieved on ClearSpeed by merging multiple kernels into one using branches and using run time variables to decide which branch of the kernel is executed, however, this is far from intuitive.

In the case of the C_N back-end that has been developed, the memory latency has been hidden using double buffering. This implementation, which is provided by the porting system, took a great deal of effort to implement and is required to make ClearSpeed function anywhere close to competitively with the GPU but is only applicable in certain circumstances.

However, ClearSpeed does provide other functionality that can be leveraged to overcome this. One of ClearSpeed's unique features is their *swizzle* operator.

Swazzle is a memory transfer between neighbouring processing elements within a ClearSpeed chip, and can provide a massive performance boost, if the application can be constructed in a manner to take advantage of it. The developed system however, is unable to leverage on the Swazzle functionality, as the information on the locality of data that is required is not expressible in C.

While this chapter has presented work that is specifically related to C_N and C for CUDA, it is clear that at the highest level the structure of both of these porting systems is similar. This means that it will be entirely possible to construct additional back-ends for other devices such as CELL, OpenCL, AMD GPUs or even multi-core CPUs using OpenMP. Doing so should only be a matter of developing the translation between the input, provided by the system client, and the target device's programming method. This is a development task of the order of several hundred programmer hours (based on personal experience), with this figure directly depending on the current level of abstraction provided by the vendor tools for the device in question. One thing that must be considered while developing back-ends for acceleration devices is that, as with ClearSpeed and Swazzle, it is not always possible for an automated porting system to leverage all of a device's functionality, especially if it is a feature unique to that device, or it requires information that is not expressible in the input language.

Chapter 5

Device Selection, Self-modification and Expandability

5.1 Introduction

One of the key abilities for an intelligent application porting system is its ability to match an application to the most appropriate acceleration device. The application classifier component of the system performs this functionality and, in essence, it functions as a black box with which other components of the system communicate.

In order to carry out this task of matching an application to a device, the application classifier stores a series of metrics and performance data for all application/device/data-set size combinations within the system.

Using the collected data, the system is then able to make an informed decision as to which device each new application should be executed on.

By the very nature of what the application classifier does, the decisions it makes

will be predictions based on previous data and, in order for the predictions it makes to be accurate, the data that they are based on must also be accurate. To ensure this, the application classifier will constantly acquire new performance data, enabling its decision making capabilities to evolve.

In short, this leads to the application classifier having three main areas of functionality:

- Storing Application Metrics.
- Decision Making.
- Acquiring new performance data to modify its classification model.

This chapter will firstly outline in detail the architecture of the application classifier and then describe how each of these three areas of functionality are implemented in the system.

5.2 Architecture of the Application Classifier

A diagram of the internal structure of the application classifier is shown in Figure 5.1 and it can be seen from this diagram that the system consists of five main components:

1. A Web Service front end.
2. A Machine Learning system powered by WEKA [63].
3. A database of metrics and performance data, stored in a MySQL database. A diagram of this database is shown in Figure 5.2.

4. A database of applications. Stored in the format output by the system client, prior to being ported to any specific device.

5. A database of data-sets, each associated with a specific application.

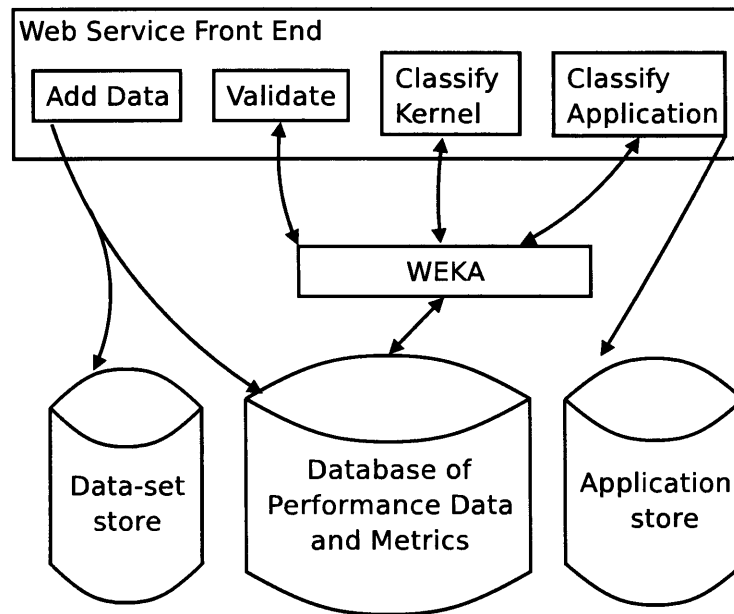


Figure 5.1: Internal Structure of Application Classifier.

The database of performance data (shown in Figure 5.2) consists of three main entities: accelerators, applications and kernels. The majority of performance data and metrics are stored in relation to kernels; each kernel will have several different entries for it showing the differing metrics for the various known problem sizes. Each entry stores the runtime and optimum device for that kernel for a specific problem size. Additionally, each application(which consists of one or more kernels) also has an optimum device associated with it for all known data-set sizes.

The database of applications and data-sets are flat file databases, consisting solely of archived copies of the application/data as appropriate, with each data-set being linked to an application ID.

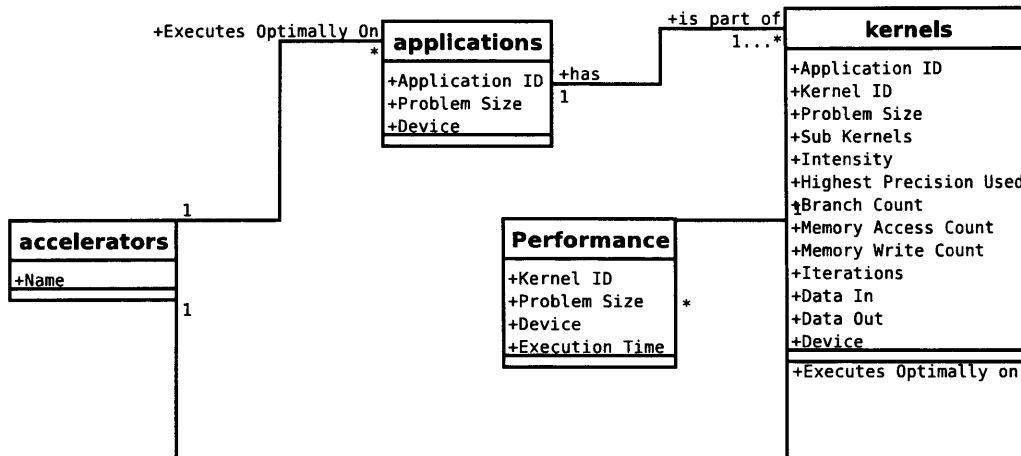


Figure 5.2: The Database of Performance Data.

These databases and the other internal components of the application classifier are only visible to the rest of the system via the four method calls provided by the web service front end.

Add Data: This method is used to add performance data to the database. It takes as input an Application ID, Kernel ID, the problem size, the recorded metrics for that problem size, the execution time as measured by the acceleration device and a copy of the data-set. This method will firstly add the execution time and metrics to the database and then, if necessary, change the device that is recorded as being optimal for this application/kernel for the problem size being executed. If a data-set is provided it will be stored in the data-set database for future use. More detail of the process of acquiring performance data is described in Section 5.6.1

Validate: This method is used to validate that the current device the application is being accelerated on is still valid. It is called when the application is executed and takes as input the application ID and problem size. It returns the optimum device for the application and problem size and stores the data-set being used. If this prediction differs from what the end user is currently accelerating the

application on, then the application may be re-reported for the new optimum device. More details on this functionality are outlined in Section 5.5

Classify Application: This method is the core of the application classifier, enabling the matching of the application to the device and takes as input the application being considered. This method first stores the application in the application database and then, using the metrics produced by the client, it returns a prediction of the optimum device for the application. This process is described in more detail in Section 5.4.

Classify Kernel: This method is a specialisation of the classify program method. It is only used to determine, by a device, if a specific kernel should be executed or not. It takes as input an Application ID, a Kernel ID, problem size, and the name of the current device. This method, using similar steps to those discussed previously, will simply determine if the kernel should be accelerated by the current device or not.

5.3 Gathering Metrics

In order for the system to make decisions a series of metrics are used. The following metrics are collected for each kernel in the input application at compile time:

- The highest precision data-type that is used by the application.
- A count of mathematical operations(Intensity).
- A count of the number of memory accesses (read and write).

- A count of branching that occurs.
- The number of iterations of the kernel that are performed.
- Size of data that must be loaded to/from the device.

These metrics can be classified into two types: static, where the value does not depend on the data-set size i.e. highest precision, and dynamic, where the value does depend on the problem size.

Static metrics can be extracted at compile time, while, in order to discover dynamic metrics such as the number of iterations and data transfer sizes, the application is instrumented and executed once at compile-time with the supplied data-set.

It should be noted that, at this stage, when extracting metrics for a kernel at compile time, only that kernel is considered and not any sub-kernels. It should be obvious that, as it is impossible to accelerate a kernel and not accelerate its sub-kernels, the metrics of sub-kernels must be considered but this is done at a later stage as a post-processing step.

5.4 Matching Applications to Devices

The process of matching the input application to a device consists of several steps:

Step 1: The application is checked to see if it is already known to the system. For an application to be known each of its kernels and the grouping of these kernels into an application must already be present within the database of performance

data. If the application is known, the matcher simply returns the optimum device stored for that application and no further steps are taken.

Step 2: An application ID is generated for the application.

Step 3: The application is checked to see if any of its kernels are already known by the system. If any of its kernels are *exactly* equal to any existing kernels in the performance database then these are duplicated and associated with the new application ID (In reality this is unlikely to occur).

Step 4: All kernels of the application that are not already present within the database are added for the initial problem size but with no performance data, or optimum device attached.

Step 5: The best possible device for each new kernel within the application is selected using WEKA [63]. This stage is, in essence, a classification problem [128] which a decision tree is utilised to solve. More detail on this phase can be found in Section 5.4.1.

Step 6: The best device for the application as a whole is selected. This is done by selecting the device which is optimum for the majority of kernels in the application weighted by the size of the kernel which is calculated as $NoIterations * Intensity$. However, it should be noted that the CPU is here treated as a special case i.e. the only time the CPU will be selected as the optimum device is if no kernels in the application provide acceleration.

Step 7: A copy of the application code is stored.

The results of this process is a recommendation for the best device for the application based on the initial problem size that has been provided.

5.4.1 Making Decisions

In order to actually make a decision regarding which device is optimum for a particular kernel, a classification algorithm is used. The classification algorithm that has been selected for use in the system is a decision tree. This algorithm was selected for the following reasons [9]:

1. A decision tree is a representation that is human-readable and easily interpreted.
2. The decision tree algorithm is a well understood algorithm allowing manual validation of the decisions the system is making.
3. A decision tree copes well with both categorical and continuous data, both of which are gathered as part of the metrics which are used by the system.
4. Decision trees are able to cope well with the presence of useless variables within the data-set. This is a situation that may arise if some of the metrics that have been used prove to be irrelevant for the performance data gathered.
5. Decision trees make no assumptions in regards to the distribution of the training data.

Within this decision tree, leaf(terminal) nodes are used to represent acceleration devices, while the non-terminal nodes are test conditions based on each of the metrics which are extracted from the kernel source code. An example decision tree is shown, for illustration purposes only, in Figure 5.3

In order to implement such a decision tree within the system, the WEKA machine learning toolkit was used. The WEKA toolkit provides several different methods

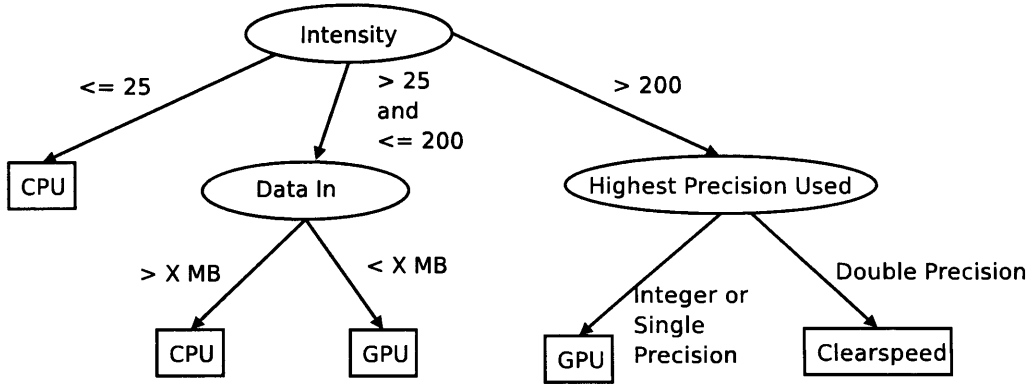


Figure 5.3: An Example Decision Tree.

of constructing decision trees, but the one that was selected is a method of decision tree induction known as *C4.5*. The *C4.5* decision tree, and its successor, *C5.0*, developed by J. Ross Quinlan have become the industry standard for decision tree induction [133] and were the obvious choice. The precise implementation of the *C4.5* algorithm that is used is known as J48 and for the purposes of our system, the suggested default parameters were used.

In order to construct the training data from data in the database a simple transformation is used to produce a kernel instance for each entry in the kernels table, excluding any that do not yet have an optimal device assigned. An illustration of such an instance is shown in Table 5.1, in addition to the data shown in the diagram it should be noted that instances are identified by an application id, kernel id and a problem size, although these values are not used when constructing the classification model. Example of these constructed instances for one of the example application are shown in Appendix B.

Intensity	Highest Precision	Branching	Memory Access	Memory Write	Iterations	Data Moved	Device
-----------	-------------------	-----------	---------------	--------------	------------	------------	--------

Table 5.1: An Kernel Instance used for Constructing a Decision Tree

The majority of items here are extracted directly from the database, however:

Intensity, Branching, Memory Write and Memory Access need to be computed to take into account the execution of subkernels. This is done in the following manner for a kernel k with a set of sub-kernels I :

$$InstanceIntensity_k = Intensity_k + \sum_{i \in I} Iterations_i * Intensity_i$$

It should also be noted that the Data Moved metric is the sum of the data that are loaded to the device and the data that are loaded back to the host from the device. The reason that the sum of these two figures are used to build the decision is that the relative sizes of the Data In and Data Out figures make implications about characteristics of an application. i.e. Reduction will produce much less output data than input data. However, these implications are better measured directly through metrics such as the intensity, number of memory reads and number of memory writes, rather than being implied by the amount of data loaded to/from the device.

The J48 decision tree, as implemented by the WEKA toolkit, is a non incremental classifier, meaning that it must be reconstructed from these instances each time new data are added. The computational complexity for constructing a decision tree (assuming a constant number of attributes) with n training instances is [133]:

$$O(n \log n) + O(n(\log n)^2)$$

This is the worst case complexity, assuming that n instances generate a tree consisting of n leaves. However, in the vast majority of cases the complexity for constructing the tree is far lower. In our results presented in Figure 6.18, the tree was constructed from 200 training instances but only has 12 leaves. This means that in many cases the computational complexity of constructing a decision tree will tend towards $O(n)$.

5.5 Validating Decisions

The previous sections have shown how the system is able to make an initial decision about which acceleration device to execute an application on based on the initial data-set that has been provided. However, the metrics of the vast majority of applications are in some way dependent on the size of the problem that they are solving. This leads to the strong possibility that the device that is favourable for one data-set size may not be the optimum for another. In fact it is widely assumed that for smaller data-set sizes the CPU will always be the preferable device.

This problem resulted in the development of the idea of “validating“ the initial selection each time the application is executed. However, in order to retain the performance advantages of executing on an acceleration device, a full analysis of the application is not practical each time it is executed.

The solution that has been used is to make use of metrics of other problem sizes for this application to compute the *estimated* metrics. Details of how this is undertaken for each metric is outlined below:

Highest Precision: Is fixed per application, so does not vary based on data-set size. The value from initial analysis is used.

Iterations, Data In and Data Out: These metrics will vary based on problem size in the vast majority of applications. The new metrics for each of these are computed as follows, assuming that for kernel k there are n previous metrics stored for this kernel for varying data-set sizes, although an example is given only for iterations:

$$Iterations_k = ProblemSize_k * \frac{\sum_i^n \frac{Iterations_i}{ProblemSize_i}}{n}$$

Intensity, Branching, Memory Write and Memory Access: These four metrics do not vary based on problem size. As the value collected is for one iteration of the kernel, but not its sub kernels. However, in order to accurately perform classification of a kernel, the sub kernels must be taken into account. This is done in the same manner that is described in Section 5.4.1.

Once the new kernel has been computed the method outlined in Section 5.4 is used to reclassify the application based on the new data-set size and, if the classification has changed, the application can be reported to the new device.

5.6 System Evolution

The previous sections of this chapter have all discussed using data stored in the application classifier to make predictions about the optimum device for an application at a specific data-set size. However, the decision that is made is only as good as the experimental data that is held. This shows a clear requirement for the system to have to a supply of experimental data in order to base its decisions on.

As mentioned previously in Section 5.4 when a new application is added to the database, its metrics are stored but no performance data is present, as it has not yet been executed by any devices.

The process of evolving the classification model used to make decisions is essentially a gap-filling algorithm. This process is described in Listing 5.1.

This algorithm is executed periodically and has the effect of ensuring that the performance database is as complete as possible given the current set of known applications and data-sets. This allows the system to fill gaps in its performance database i.e. if a new data-set has become available for an existing application but there are some existing devices that this particular application and data-set combination has not been executed on.

This process happens without user intervention and makes use of free runtime on the devices present within the system, as opposed to interfering with user's execution of applications. This methodology was adopted due to the possibility that the instrumentation of the application code would negatively impact the performance, something which would be undesirable to the end user.

Listing 5.1: Algorithm for Integration of New Devices

```
for each application A in applications table
  for each data-set D associated with the application
    for each accelerator C in accelerators table
      if Performance data is not available for application A on accelerator C using a
        data-set of size D
      then
        execute application A on accelerator C using data-set D and record performance
      end if
    end for
    update optimum device for application A using data-set D
  end for
end for
```

5.6.1 Gathering Performance Data

The performance data required by the system is acquired by re-reporting the application and adding generated instrumentation to the code, in order to collect the following metrics for each kernel:

- Wallclock execution time of the kernel. This is the execution time of all generated code including device initialisation, loading data to/from the device and the execution of the device program.
- Updated iteration count.
- Updated Data In and Data Out figure.

The instrumentation that is inserted consists of standard ANSI C code that outputs the metrics to a file that is then read by the system.

Updating Iteration Count: This metric is updated by outputting the number of iterations of the kernel prior to its execution starting.

Updating Data In and Data Out Count: These metrics are updated by printing to the file, the sum of the sizes of input/output data loaded to/from the device, prior to execution of the kernel.

Collecting Execution Time: The execution time is collected by inserting two pieces of code, shown in Listings 5.2 and 5.3. The code to start the timer is added immediately prior to the first line of generated code for the kernel and the timer stop code is added immediately after the final line of code in the generated kernel. This enables the system, on a per kernel basis, to measure the execution time of the kernel, including initialisation and data movement overheads in a consistent

manner. If the kernel that is being instrumented is a subkernel which may be executed many times, then the average (mean) execution time is used.

Listing 5.2: Timing Code: Starting the Timer

```
long long timer_start() {  
    struct timeval time;  
    gettimeofday(&time, NULL);  
    return (time.tv_sec * 1000000 + time.tv_usec)  
        /1000;  
}
```

Listing 5.3: Timing Code: Stopping the Timer

```
long long timer_stop( long long previousTime) {  
    return timer_start() - previousTime;  
}
```

The decision to include all of the overheads is only natural when considering that a fair comparison with CPU execution is essential; it is useless to accelerate an application, even if the performance on the device is better than on the CPU, if the total execution time, including overheads such as data transfer, is greater than if the code was executed on the CPU.

However, when performing this instrumentation, there are two special cases which must be dealt with. Firstly, the CUDA compiler performs optimisations, to reduce the memory transfer between host and device, when multiple kernels are present. So in order to gain accurate performance data of one kernel's execution, each kernel must be executed in isolation.

Secondly, the CUDA programming model dictates that the method call to execute the kernel on the device is non-blocking and that blocking occurs in the method calls to load data back onto the host [46]. This means that in cases where data

is not loaded back to the host at the end of the execution of a kernel, the timer methods will report the execution time as very low. In order to counteract this the code shown in Listing 5.4 is inserted into the generated code immediately after the kernel call. This code will block until the Kernel execution is finished, ensuring accurate timing results are obtained.

Listing 5.4: Ensuring CUDA execution finished before stopping the timer.

```
cudaThreadSynchronize ();
```

5.6.2 Integration of New Devices

As a special case of the process of acquiring performance data, the application classifier is also able to pro-actively improve its own knowledge-base by adapting to the introduction of new devices or newer versions of existing devices.

This functionality is motivated by the need for the system to be able to bootstrap any new devices that are added to the system. Without such a process, a newly added device would not be selected to execute applications as the application classification system would continually select devices that it already has performance data for. The solution that has been developed consists of two phases: firstly, the system must discover any new devices and add them to its database. Secondly, the system must integrate these devices into the system by generating performance data for them, this step is identical to the process of gap-filling that has already been discussed in this section.

In order for a new device to be added to the system a set of pre-requisites must be met:

- A back-end porting system must be developed/adapted for the device.

- A web services server must be installed on the host node that the device is attached to.
- The device must be named.

Each type of device must have a unique name within the system. The device name must also allow for differentiation between versions of the same device. i.e. if the device name GPU represented GPU model *X* then the newer model *Y* could be named GPU2.

Once these pre-requisites have been met, all that is required for the user to add the device to the system is add it to the UDDI server, this is normally as simple as adding the host nodes IP address to the UDDI server.

Once the entry has been added to the UDDI server, the application classifier (which periodically checks for new devices) will pick up that a new device is present and add it to its internal database. This step then allows the device to be fully integrated into the system using the process that has been previously discussed and the algorithm mentioned in Listing 5.1.

5.7 Chapter Summary

This chapter has described the architecture and functionality of the application classifier component of the Application Porting System. This component, although used as a service by both the system client and the individual acceleration devices, forms the core of the system.

The application classifier stores all the performance knowledge that the system has gained and uses this data to predict a-priori the optimum device for applica-

tions. It is also able to provide updates, and improve existing recommendations as new knowledge becomes available.

In order to produce these predictions the application classifier uses a classification model: a decision tree. The decision tree algorithm that is used is the J48 implementation of a C4.5 decision tree. The primary reason for choosing the decision tree algorithm, is that it is a relatively simply, but powerful classification model. It is human readable enabling the constructed tree to be analysed and, additionally, used to manually validate classifications that have been made. The J48 algorithm that has been utilised is non incremental meaning that every time the training data that built the tree is modified the tree must be rebuilt. This has the side effect of ensuring that the algorithm is deterministic, meaning that regardless of the order that training data are received by the system, the resultant tree will always be the same.

The application classifier itself consists of a web services front end, a machine learning system and a set of databases storing performance data, application code and sample data-sets.

The application classifier that has been outlined in this chapter is a key component of the novel self-modifying application porting system that is described in this thesis. It operated in both a pro-active and reactive system; responding to requests and storing performance data that is provided to it. The system will pro-actively detect and fill gaps in its performance database, allowing the classification model to be modified, thus improving the recommendations that it gives without user interaction. This process is also used to allow the automatic integration and bootstrapping of new devices into the system, once the device had been added to the UDDI server. This process is necessary because any device that the system did not possess performance data for could never be selected to execute an application.

Another possible use of the application classifier component, along with a populating classification model, could be as a stand alone recommendation service. Such a service would prove valuable to non computer science users with limited knowledge of application acceleration by providing a prediction of the optimum device for an application(s) prior to the purchase of hardware, in order to ensure that the correct device is purchased.

Chapter 6

Application Case Studies

6.1 Introduction

In order to evaluate the performance of the system that has been built, a series of test applications were run through the system. These test applications aim to show that the system is functioning according to the following goals:

1. The system is generating device specific code which performs comparably to hand ported code.
2. The system is able to select the most appropriate device based on performance data it holds.
3. The system is able to modify its internal classification model based on the data acquired.

In order to fully evaluate the system, a set of test examples were selected. However, before these examples were executed, a set of simpler examples were first ported in order to provide the system with a base set of performance data that it will use to make predictions on the unseen test examples that will later be trialled with the system.

For each application selected, both seed and test, a single precision and a double precision version will be executed and, for both of these versions, the following results will be presented for a variety of data-set sizes:

- Wall-clock execution times of the application running on the CPU, GPU and ClearSpeed.
- Wall-clock execution times of a hand ported version of the application running on the same hardware.
- A comparison of these performances relative to the CPU control device.
- The performance difference between the optimised hand ported code, produced by the developer, and automatically ported code, produced by the system.
- Peak performance, measured in GFlop/s, and the application's memory bandwidth, measured in GB/second, will be presented to enable a comparison to any re-factored or tuned versions of the applications that exist.

In the results tables presented in this chapter, it should be noted that where **X** is shown this means that the execution time for that kernel/application could not be measured, this means that either the application was not executable at that data-set size, the kernel itself was not executable or the kernel execution time was too long to measure (this time-out is a set parameter).

It should be noted that these performance results are taken from the results acquired by the acceleration back-end devices (Discussed in Chapter 5) and an overall measure of each applications wall-clock runtime. This means that for the measured execution times of the kernels, the runtime acquired includes the device initialisation, kernel runtime and memory transfers. The overall application

runtime is the entire execution time of the application on the node hosting the acceleration device. This includes all non-accelerated code running on the CPU.

Additionally, the device code generated by the human programmer is a direct port from the CPU code to the respective devices. In many cases, it is possible to get higher performance by re-factoring the input code or adapting it to utilise libraries available for the device. These cases are shown by way of comparisons to re-factored applications which illustrate the performance that can be achieved given sufficient development time and the availability of expert developers. However, with regards to the hand ported versions of the applications, the purpose of the comparison is to confirm that the code generators are producing efficient direct ports without re-factoring. As such, these hand-ports have been developed with rigour and all reasonable methods for improving performance have been taken short of undertaking re-factoring of the original algorithm.

The peak performance figures that have been presented for each application are calculated from the metrics stored within the Application Porting System. Computation is calculated as follows where *KernelExecutionTime* is execution time excluding the time taken for transfer of data to the device, and *NoFloatingPointOperationsPerKernel* is the number of flops per iteration of the kernel:

$$Computation = \frac{NoFloatingPointOperationsPerKernel * NoIterations}{KernelExecutionTime}$$

Memory Bandwidth is calculated as follows, where *SizeDataTransfer* is the number of bytes that is moved between the processing element and device's main memory:

$$\text{MemoryBandwidth} = \frac{(\text{SizeOfDataTransfer}) * \text{NoIterations}}{\text{KernelExecutionTime}}$$

Finally, transfer rate is calculated as:

$$\text{TransferRate} = \frac{\text{SizeOfDataTransferred}}{\text{MemoryTransferTime}}$$

However, there are slight issues in measuring some of the inputs for these equations. The Application Porting system measures the execution time of each kernel(excluding memory transfer time) and the total execution time(including time taken to transfer data to the device). However, when measuring the time taken for memory transfers to the GPU it is reliant on the data given by the CUDA profiler (for ClearSpeed this was measured directly).

In addition to the performance data described above, for the more complex test examples, detailed analysis will be presented on the decisions that the system made when predicting the optimum device for the application. Additionally, the decisions will be validated to determine if the chosen device is actually the optimum device for the application.

The applications that were selected as seed applications, were selected as “building block” applications based on their categorisation within the Seven Dwarfs of Applications, these Dwarfs, which were first described by Asanovic et al, constitute classes of applications defined by similarity in computation and data movements [14]:

- Dense Linear Algebra,
- Sparse Linear Algebra,

-
- Spectral Methods,
 - N-Body Methods,
 - Structured Grids,
 - Unstructured Grids,
 - Monte Carlo.

From these categories the following applications were selected as seed applications:

- Dense Linear Algebra - GEMM,
- Structured Grids - Sobel Edge Detector,
- An N-Body Simulation,
- A Monte Carlo Simulation.

The three more complex applications that were selected as test applications are:

- A 2D Fast Fourier Transform,
- A Canny Edge Detector,
- An iterative ray tracer.

All of the experiments discussed in this chapter were conducted on the same system. This system consists of two acceleration devices: A NVIDIA Tesla

C1060 GPU with 4GB of GPU memory running CUDA version 2.2 and a single ClearSpeed e710 accelerator board, consisting of two CSX600 ClearSpeed Chips with 1GB of shared memory running version 3.1 of the C_N SDK. Acting as a control for the gathered performance data is an Intel Xeon 3.0GHz with 16GB RAM, with data shown for both single-core and estimated quad-core performance of this chip. All CPU code is compiled using GCC version 4.1.2. For comparison the optimum performance characteristics of both devices are shown in Table 6.1. Please note that the quoted performance for ClearSpeed is for one chip, so the theoretical maximum computation achievable for the board that is utilised in this chapter will be twice this figure.

Device	Computation GFlop/s	Memory Bandwidth GB/s
Single Precision		
GPU	933	102
ClearSpeed	25	96
Double Precision		
GPU	78	102
ClearSpeed	25	96

Table 6.1: Optimal Performance Characteristics of Devices[43][36]

In all the results shown in this chapter estimated quad-core performance is used. The estimated quad-core CPU was used in order to provide a comparison to the best possible single chip CPU performance that is currently available. In order to achieve this “best possible” comparison it is assumed that all kernels encountered give a four times performance improvement when running on a quad-core CPU. This is an assumption that linear speedup is achievable and, to verify that it is a reasonable assumption, a series of experiments were conducted with a subset of the applications discussed in this chapter ported to utilise all four cores of a Intel Xeon 3.0GHz, using GCC’s inbuilt OpenMP support. These results are shown in Table 6.2 and show that in both precisions an average performance

improvement of $\approx 2.9x$ has been achieved. These results suggest that while employing a quad-core CPU will not provide a 4x performance improvement, it is a useful approximation to enable the simulation of a “best possible” multi-core version of the CPU that is being used and could be extended for future generations of CPUs i.e. oct-core.

Application	Execution Time /s								
	N-Body	Sobel	Canny				FFT		
Kernel	1	2	1	2	3	4	1	2	3
Single Precision									
Single Core	2.07	4.90	3.55	6.57	1.22	0.17	14.85	7.53	1.45
Quad Core	0.68	1.54	1.28	2.12	0.42	0.05	5.31	4.57	0.52
Speedup	3.04x	3.19x	2.77x	3.10x	2.91x	3.53x	2.80x	1.65x	2.79x
Double Precision									
Single Core	2.75	5.00	3.67	6.62	1.27	0.22	15.86	7.75	2.1
Quad Core	0.89	1.58	1.33	2.14	0.43	0.09	5.58	3.78	0.61
Speedup	3.09x	3.17x	2.76x	3.09x	2.96x	2.45x	2.84x	2.05x	3.45x

Table 6.2: OpenMP Performance

6.2 Seed Applications

In addition to the four specified seed applications two even simpler applications were also run through the system. These applications were deliberately selected as some of the simplest pieces of code that could be constructed but still be parallelisable. The two applications that were trialled were:

- Zeroing memory.
- Addition of two arrays.

Both of these two applications were executed using data-set sizes from 640,000 to 25,000,000 data-items in both single and double precision and in all cases, due to the lack of any mathematical complexity, the CPU offered the best performance.

6.2.1 Structured Grids - Sobel Edge Detector

The Sobel Edge Detector application involves finding the magnitude of the gradient of each pixel in a 2D black and white image. This is done by convolving two 3x3 masks (A and B), which are shown in Figure 6.3, over the image. The magnitude of the gradient can then be calculated for each pixel by:

$$|G| = \sqrt{A^2 + B^2}$$

-1	0	+1	+1	+2	+1
-2	0	+2	0	0	0
-1	0	+1	-1	-2	-1

Table 6.3: Sobel Convolution Masks A and B

Data Size n	Execution Time (Seconds)							
	CPU	Quad Core CPU	GPU			ClearSpeed		
			Human Port	System Port	% Diff	Human Port	System Port	% Diff
1000x1000	0.16	0.12	1.06	1.15	8%	0.22	0.28	31%
2000x2000	0.41	0.26	1.21	1.27	4%	0.64	0.68	5%
4000x4000	1.43	0.84	1.78	1.91	7%	2.36	2.53	7%
6000x6000	3.19	1.86	2.69	3.01	12%	5.2	5.48	5%
8000x8000	5.63	3.27	3.69	4.46	21%	9.04	9.57	6%
10000x10000	8.65	4.98	5.47	6.49	19%	✗	✗	✗

Table 6.4: Single Precision Execution Times for Structured Grids Application (✗ signifies that the application failed to execute)

Data Size n	Execution Time (Seconds)							
	CPU	Quad Core CPU	GPU			ClearSpeed		
			Human Port	System Port	% Diff	Human Port	System Port	% Diff
1000x1000	0.14	0.10	1.09	1.17	7%	0.25	0.25	2%
2000x2000	0.41	0.26	1.26	1.34	7%	0.75	0.79	6%
4000x4000	1.54	0.93	1.9	2.17	14%	2.7	2.84	5%
6000x6000	3.38	2.02	2.97	3.42	15%	5.94	6.22	5%
8000x8000	5.96	3.55	4.48	5.29	18%	×	×	×
10000x10000	9.27	5.52	6.37	7.73	21%	×	×	×

Table 6.5: Double Precision Execution Times for Structured Grids Application

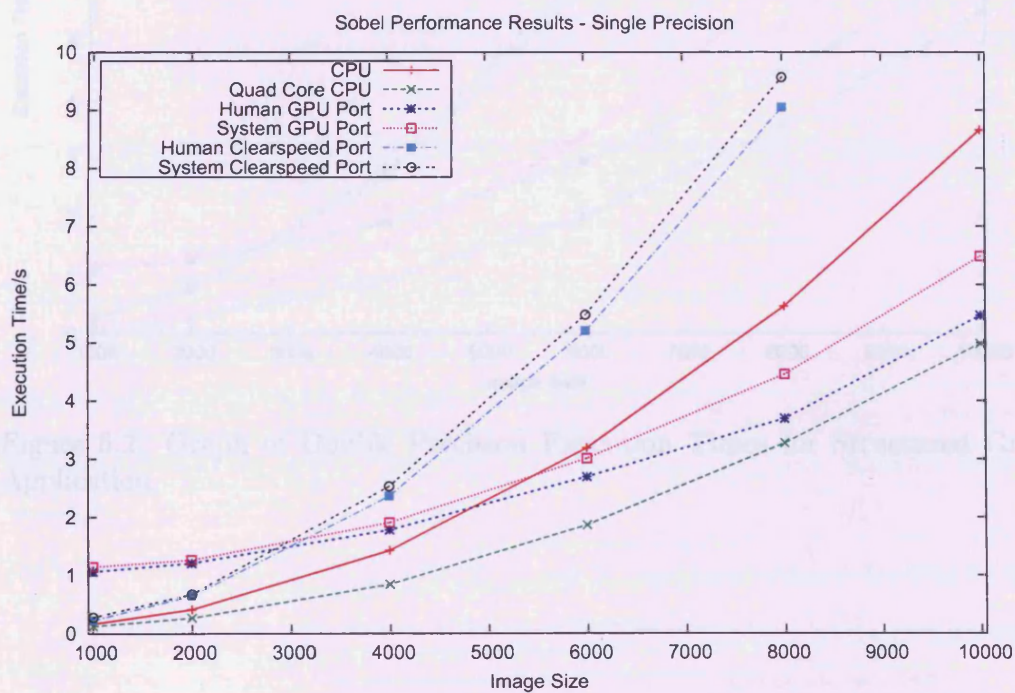


Figure 6.1: Graph of Single Precision Execution Times for Structured Grids Application.

Data Size	Single Precision Performance			Double Precision Performance		
	CPU	Clearspeed	Optimal Device	GPU	Clearspeed	Optimal Device
1000x1000	0.17s	0.15s	GPU	0.07s	0.05s	GPU
2000x2000	0.35s	0.30s	GPU	0.10s	0.08s	GPU
4000x4000	0.65s	0.55s	GPU	0.14s	0.11s	GPU
6000x6000	1.02s	0.85s	GPU	0.20s	0.16s	GPU
8000x8000	1.50s	1.25s	GPU	0.28s	X	GPU
10000x10000	2.10s	X	GPU	0.41s	X	GPU

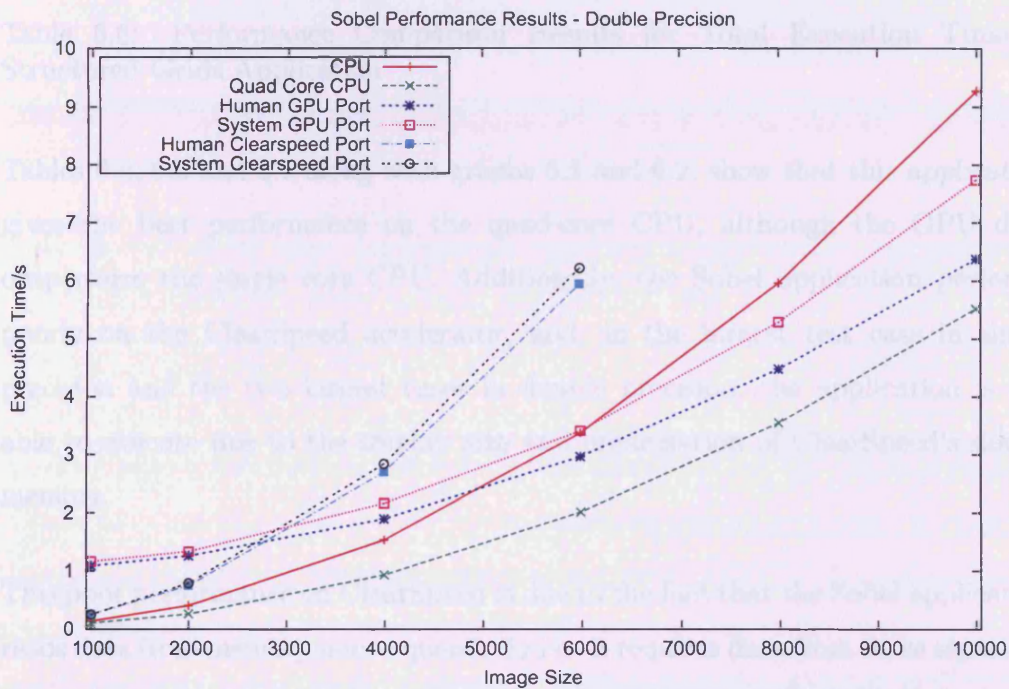


Figure 6.2: Graph of Double Precision Execution Times for Structured Grids Application.

Data Size n	Single Precision Performance			Double Precision Performance		
	GPU	ClearSpeed	Optimal Device	GPU	ClearSpeed	Optimal Device
1000x1000	0.10x	0.43x	QCPU	0.01x	0.40x	QCPU
2000x2000	0.20x	0.38x	QCPU	0.19x	0.33x	QCPU
4000x4000	0.43x	0.33x	QCPU	0.43x	0.33x	QCPU
6000x6000	0.62x	0.34x	QCPU	0.59x	0.32x	QCPU
8000x8000	0.89x	0.34x	QCPU	0.67x	✗	QCPU
10000x10000	0.77x	✗	QCPU	0.71x	✗	QCPU

Table 6.6: Performance Comparison Results for Total Execution Time of Structured Grids Application

Tables 6.4, 6.5 and 6.6 along with graphs 6.1 and 6.2, show that this application gives the best performance on the quad-core CPU, although the GPU does outperform the single core CPU. Additionally, the Sobel application performs poorly on the ClearSpeed accelerator, and, in the largest test case in single precision and the two largest cases in double precision, the application is not able to execute due to the smaller size and organisation of ClearSpeed's device memory.

This poor performance on ClearSpeed is due to the fact that the Sobel application reads data from memory non-sequentially i.e. it requires data from three separate rows of the source image in order to compute the current pixel. This means that the ClearSpeed device must load the data from the device memory into the processing element memory in smaller chunks, and, unlike CUDA, ClearSpeed has no method of hiding this memory latency. It is however, interesting to note that for smaller data-set sizes ClearSpeed outperforms the GPU, but, at these data-set sizes, the CPU performs better than both acceleration devices.

When comparing the performance of the manual and automatic ports, we can see that differences (baring the smallest ClearSpeed test case) are all within $\approx 21\%$ of

the hand ported code. The main reason for these differences is that the manual CUDA port is able to take advantage of loading some items of memory into the multiprocessor's shared memory in order to reduce the amount of memory transfers needed.

Comparison with Existing Work

While no direct performance data from other implementations of the same application could be found, there are several examples in literature of similar applications running on ClearSpeed devices and GPUs.

Heuveline et al[65] have implemented the Lattice Boltzmann Method on a ClearSpeed accelerator card utilising the CSX600 chip (the predecessor to the chip used in this thesis). While this application is radically more complex than the Sobel implemented here it is also categorised into the Structured Grids dwarf[14]. In their report the authors implement the algorithm on the device and then compared it against a Xeon CPU, before deciding that only a small part of the overall algorithm (the collision step) was actually beneficial to implement on the Accelerator board. However, the performance results for this part of the application shows that they have achieved a 1.6x speed-up compared to the CPU.

Examining GPU based applications, Brandvik et al[21] in 2009 developed TurboStream, a Navier Stokes solver. In this paper the author has provided performance figures that compare a single NVIDIA GT200 GPU and a quad-core Xeon 2.33GHz Processor. This data shows that the NVIDIA GPU provides a performance improvement of approximately 20x, reducing the computation time to 1 minute on the GPU from 20 minutes on the CPU. In a second publication[22], Brandvik et al compare the performance of generated CUDA code from a source code generator to that of a CPU. The code generator that the author has

developed takes as input the Python definition of a stencil kernel and produces the code for an equivalent CUDA kernel. The generated GPU code provides, on average, 300-440% performance improvement over the CPU and the author calculates that the GPU achieves between 13 and 99 GFlop/s performance and between 18 and 47 GB/s memory bandwidth.

When comparing these performance figures to those for the Sobel shown in Table 6.7 it can be seen that the performance of the automatically generated code only achieves approximately 2.5 GFlop/s. This data also shows that the memory bandwidth achieved within the kernel is very low. This low memory bandwidth is the main factor preventing a further performance improvement. In order to improve the performance additional work will need to be done to increase the memory efficiency of the generated kernel, possibly by utilising CUDA's shared memory functionality or by reducing the number of memory reads by changing the way in which the kernel accesses the device memory.

Device	Peak Performance GFlop/s	Peak Memory Bandwidth GB/s	Peak Transfer Rate GB/s
Single Precision			
GPU	2.58	1.86	1.19
ClearSpeed	1.21	0.87	0.34
Double Precision			
GPU	2.62	3.78	1.33
ClearSpeed	1.1	1.59	0.39

Table 6.7: Sobel Application Peak Performance

6.2.2 Dense Linear Algebra - Matrix Multiplication

The next application that has been executed using this system was a general matrix multiplication application. This application utilised the standard GEMM

formula $C = \alpha AB + \beta C$, where the new matrix C is computed based on the product of two matrices A and B and the old matrix C . The dataset size n signifies that the size of matrices A , B and C is $n * n$.

Data Size n	Execution Time (Seconds)							
	CPU	Quad Core CPU	GPU			ClearSpeed		
			Human Port	System Port	% Diff	Human Port	System Port	% Diff
200	0.15	0.12	1.13	1.14	1%	0.18	0.23	28%
800	4.00	1.93	2.55	2.59	2%	2.70	2.72	1%
2000	51.72	18.68	15.73	18.24	16%	25.46	25.65	1%
2800	135.97	45.33	40.00	40.86	2%	72.92	73.21	1%
4000	382.84	118.60	110.62	133.57	21%	166.86	167.39	1%
5000	738.06	220.07	224.65	249.65	11%	337.59	339.05	1%

Table 6.8: Single Precision Execution Times for GEMM Application

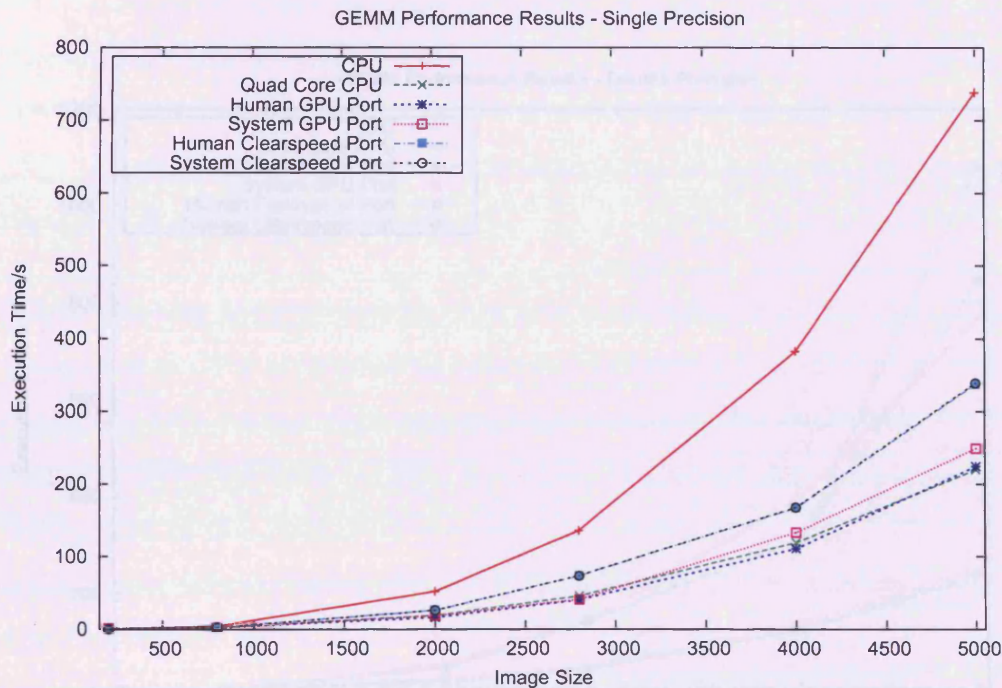


Figure 6.3: Graph of Single Precision Execution Times for GEMM Application.

Data Size n	Execution Time (Seconds)							
	CPU	Quad Core CPU	GPU			ClearSpeed		
			Human Port	System Port	% Diff	Human Port	System Port	% Diff
200	0.18	0.15	1.10	1.25	14%	0.16	0.19	19%
800	4.13	2.03	2.86	2.87	1%	3.89	3.93	1%
2000	53.68	19.70	17.98	22.05	23%	46.01	46.40	1%
2800	142.07	47.90	41.60	42.63	3%	142.28	143.12	1%
4000	399.56	125.11	124.22	137.82	11%	311.90	312.61	1%
5000	768.22	232.23	242.75	255.76	6%	842.67	1084.55	29%

Table 6.9: Double Precision Execution Times for GEMM Application

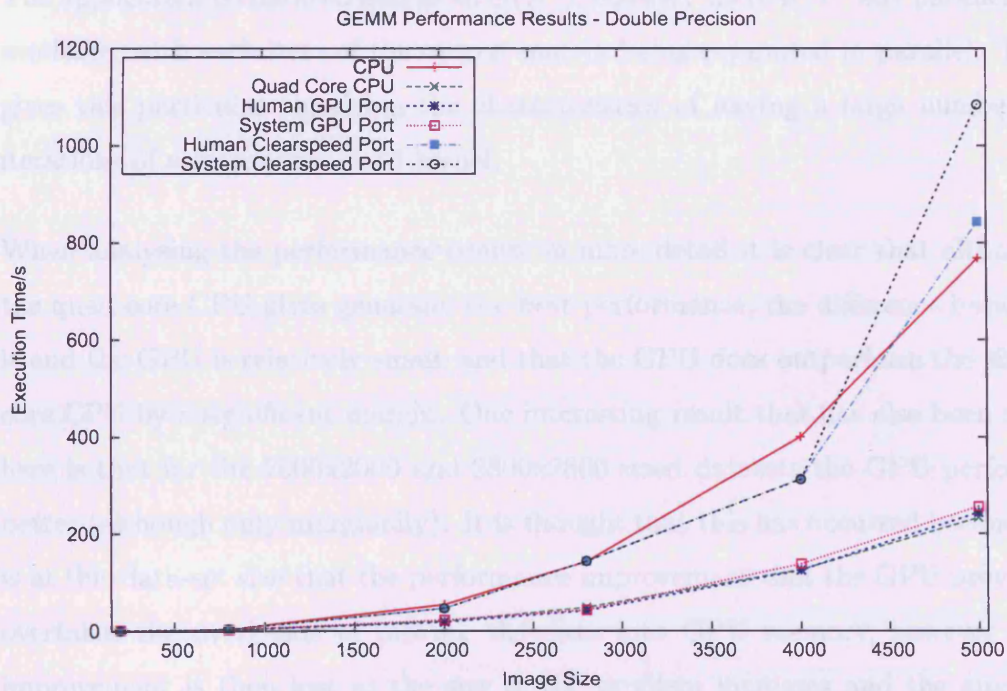


Figure 6.4: Graph of Double Precision Execution Times for GEMM Application.

Data Size n	Single Precision Performance			Double Precision Performance		
	GPU	ClearSpeed	Optimal Device	GPU	ClearSpeed	Optimal Device
200	0.11x	0.52x	QCPU	0.12x	0.79x	QCPU
800	0.75x	0.71x	QCPU	0.70x	0.52x	QCPU
2000	1.02x	0.73x	GPU	0.88x	0.42x	QCPU
2800	1.11x	0.62x	GPU	1.12x	0.33x	GPU
4000	0.89x	0.71x	QCPU	0.91x	0.40x	QCPU
5000	0.89x	0.65x	QCPU	0.91x	0.21x	QCPU

Table 6.10: Performance Comparison Results for GEMM Application

As can be seen from the performance results presented in Tables 6.8, 6.9 and 6.10 and Figures 6.3 and 6.4, the matrix multiplication largely gives the best performance on the quad core CPU.

The application is characterised as an $O(N^3)$, however there is N^2 way parallelism available, with each item of the output matrix being computed in parallel. This gives this particular algorithm the characteristics of having a large number of iterations of a reasonably sized kernel.

When analysing the performance results in more detail it is clear that although the quad core CPU gives generally the best performance, the difference between it and the GPU is relatively small, and that the GPU does outperform the single core CPU by a significant margin. One interesting result that has also been seen here is that for the 2000x2000 and 2800x2800 sized datasets the GPU performs better (although only marginally). It is thought that this has occurred because it is at this data-set size that the performance improvement that the GPU provides overtakes the overheads of moving the data into GPU memory, however this improvement is then lost as the size of the problem increases and the amount transfers to/from the GPU's core to its main memory also increases.

Another key point to note is that the performance difference between the

automatic and manual port is small compared to the data-set size, this causes the percentage difference between the two ports to vary quite considerably, but it is encouraging to note that, except for examples with very small run times and the largest ClearSpeed experiment in double precision, the difference between hand ported and automatically ported code is $\approx 20\%$.

When considering the performance of the ClearSpeed accelerator: this particular application outperforms the single core CPU in single precision. However, in double precision, ClearSpeed only slightly outperforms or matches the CPU's performance and for the final dataset the ClearSpeed performance is noticeably worse. This is due to ClearSpeed's memory management model, as the 5000×5000 matrix is too large for each chip to store a copy in its own local memory the chips must start sharing memory between them, this incurs a significant performance overhead as shown by the results.

Comparison with Existing Work

Dense linear algebra, under which DGEMM is classified are one of the application types that are most commonly ported to an acceleration device. In terms of ClearSpeed, the manufacturer provides an extensive BLAS(Basic Linear Algebra Sub-programs) library, and McIntosh-Smith claimed in a presentation that executing their DGEMM on the most recent model achieved performance approaching the chip's maximum of 25GFlop/s[88].

On the GPU, a great deal of work has been done in optimising BLAS routines and CUDA have provided their own BLAS library, which includes an implementation of DGEMM[103] shown to achieve, on a current TESLA GPU, a maximum of approximately 350GFlop/s in single precision and up to approximately 70GFlop/s in double precision[96]. As discussed in Section 2.6, this performance is however

heavily dependant on the dimensions of the matrix being considered. Finally Nath et al have shown that on the NVIDIA FERMI M2050 (a higher performance GPU than used in this thesis) that they can achieved performance of up to 300GFlop/s in double precision, this performance was achieved using the MAGMA BLAS DGEMM[97].

Comparing these results to the performance shown in Table 6.11 we can see that the automatically generated code is nowhere near achieving the peak performance discussed above. This is simply because of the issues with directly porting the standard GEMM algorithm. In order to improve performance the GEMM algorithm needs to decompose the problem, this is not exhibited in input CPU code, so it is not something that can be automatically generated using the methods employed in this thesis. A second related problem is that when running on almost any processor the GEMM algorithm is memory bound, so in order to provide improved performance on the GPU far more work is needed to ensure that the maximum possible memory bandwidth is achieved, such as ensuring the memory access is coalesced and making extensive use of the GPU's shared memory.

Device	Peak Performance GFlop/s	Peak Memory Bandwidth MB/s	Peak Transfer Rate MB/s
Single Precision			
GPU	3.70	4.94	1.6
ClearSpeed	2.59	3.46	0.32
Double Precision			
GPU	3.62	9.83	2.00
ClearSpeed	0.73	1.94	0.59

Table 6.11: GEMM Application Peak Performance

6.2.3 N Body Methods

The next seed example that was executed by the system is an N-Body simulation based on the all-pairs method outlined in [32], where the initial inputs to the problem are a set of n bodies $b_1 \dots b_n$ each body i has mass m_i , velocity v_i and position p_i . The distance between any two bodies is written d_{ij} and the force on body i due to body j is written f_{ij} .

The algorithm then carries out the following steps

- Compute f_{ij} for all pairs of bodies. $f_{ij} = \frac{Gm_i m_j r_{ij}}{|r_{ij}|^3}$ where $i \neq j$
- Compute total force on each body $f_i = \sum_{i,j \neq i} f_{ij}$
- Update the position p_i and velocity v_i of each body $p_i = p_i + v_i \Delta t + \frac{\Delta v_i}{2} \Delta t^2$ and $v_i = v_i + \frac{f_i \Delta t}{m_i}$

Data Size n	Execution Time (Seconds)							
	CPU	Quad Core CPU	GPU			ClearSpeed		
			Human Port	System Port	% Diff	Human Port	System Port	% Diff
500	0.82	0.80	1.08	1.13	5%	0.77	2.67	246%
1000	3.24	0.84	1.17	1.19	2%	2.01	4.79	138%
2000	12.94	3.27	1.49	1.60	7%	6.64	13.88	109%
4000	51.67	13.00	2.23	2.39	7%	24.64	50.48	105%
6000	116.21	29.31	2.49	2.59	4%	55.98	111.15	99%
8000	206.53	52.26	3.05	3.15	3%	97.85	196.05	100%

Table 6.12: Single Precision Execution Times for N-Body Application

Data Size n	Execution Time (Seconds)							
	CPU	Quad Core CPU	GPU			ClearSpeed		
			Human Port	System Port	% Diff	Human Port	System Port	% Diff
500	1.04	0.92	1.36	1.39	2%	1.01	3.14	211%
1000	4.14	1.01	1.75	1.88	7%	3.52	7.74	120%
2000	16.71	3.93	5.09	2.63	4%	12.27	25.02	104%
4000	66.61	17.46	5.56	5.63	1%	46.18	94.78	105%
6000	150.016	39.73	10.51	10.53	1%	105.41	211.11	101%
8000	268.02	68.75	13.78	13.86	1%	184.30	373.85	103%

Table 6.13: Double Precision Execution Times for N-Body Application

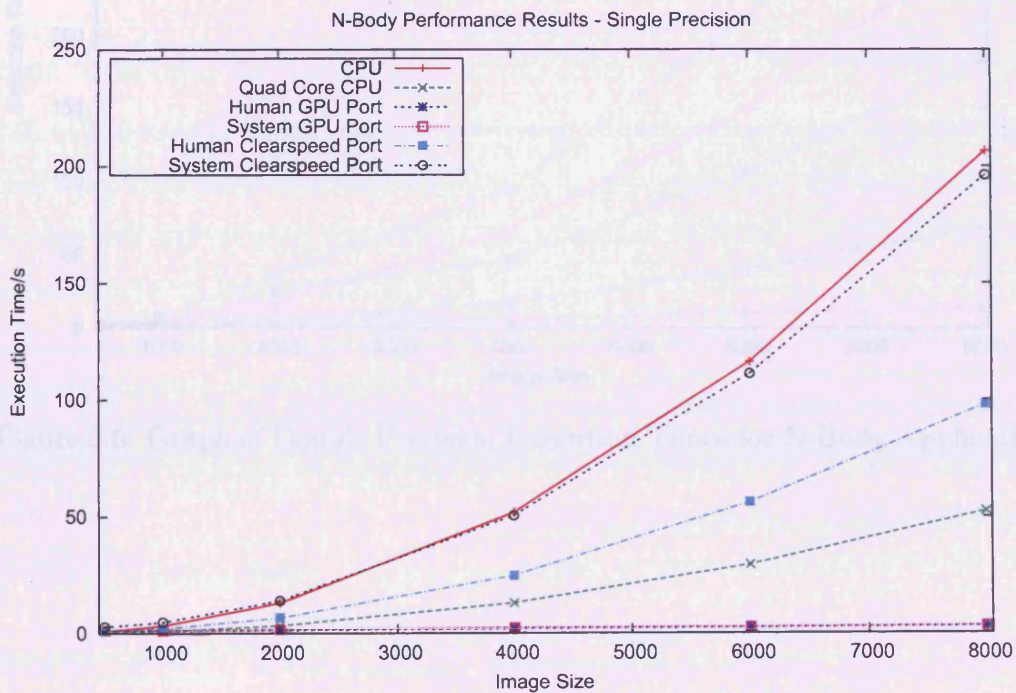


Figure 6.5: Graph of Single Precision Execution Times for N-Body Application.

No. Bodies	Single Precision Performance			Double Precision Performance		
	GPU	Clearspeed	Optimal Device	CPU	Clearspeed	Optimal Device
100	0.71s	0.74s	GPU	0.66s	0.39s	GPU
1000	0.71s	0.75s	GPU	0.52s	0.34s	GPU
10000	2.94s	0.94s	GPU	1.68s	0.44s	GPU
100000	5.44s	0.76s	GPU	4.17s	0.41s	GPU
1000000	11.64s	0.77s	GPU	12.43s	0.41s	GPU

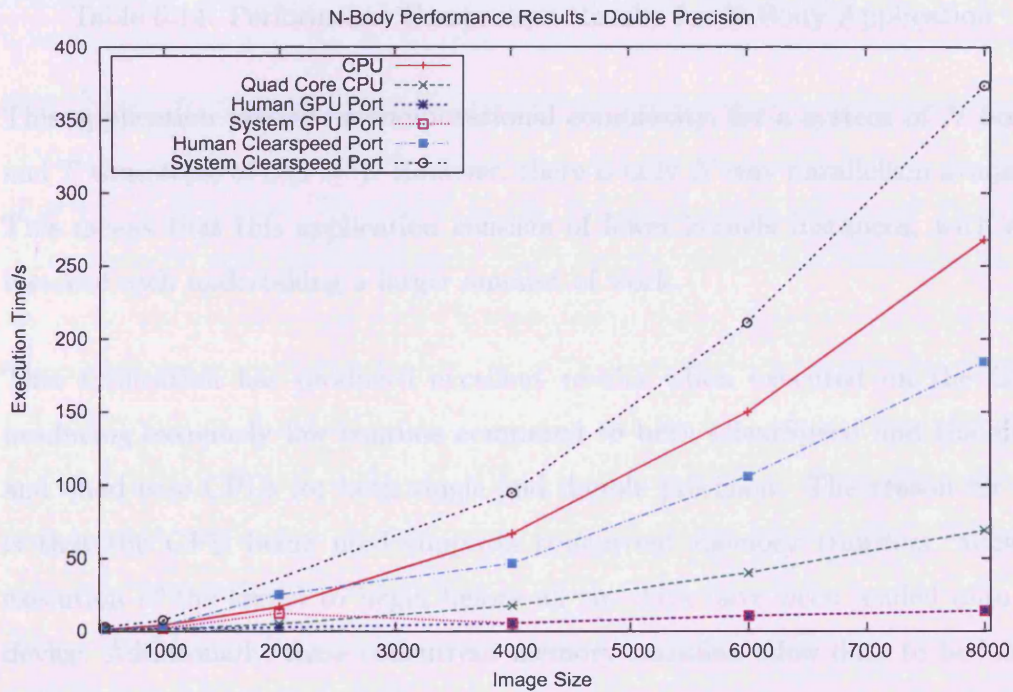


Figure 6.6: Graph of Double Precision Execution Times for N-Body Application.

No Bodies	Single Precision Performance			Double Precision Performance		
	GPU	ClearSpeed	Optimal Device	GPU	ClearSpeed	Optimal Device
500	0.71x	0.30x	QCPU	0.66x	0.29x	QCPU
1000	0.71x	0.18x	QCPU	0.54x	0.13x	QCPU
2000	2.04x	0.24x	GPU	1.49x	0.16x	GPU
4000	5.44x	0.26x	GPU	3.10x	0.18x	GPU
6000	11.32x	0.26x	GPU	3.77x	0.19x	GPU
8000	16.60x	0.27x	GPU	4.96x	0.18x	GPU

Table 6.14: Performance Comparison Results for N-Body Application

This application presents a computational complexity, for a system of N bodies and T timesteps, of $O(TN^2)$. However, there is only N way parallelism available. This means that this application consists of fewer kernels instances, with each instance each undertaking a larger amount of work.

This application has produced excellent results when executed on the GPU, producing extremely low runtime compared to both ClearSpeed and the single and quad core CPUs for both single and double precision. The reason for this is that the GPU being used supports concurrent memory transfers, allowing execution of the kernel to begin before all the data have been loaded onto the device. Additionally, these concurrent memory transfers allow data to be loaded back from the device before the execution of the current kernel has finished. In order to leverage this functionality, no code modifications are necessary, the only requirement is that the GPU is of a sufficiently recently model and supports the concurrent functionality.

In single precision and for the hand ported code in double precision, ClearSpeed produces performance better than that of the single core CPU. It is, however, significantly lower than that of the GPU. This is due to the fact that the N-body application needs a relatively large amount of data to be kept in the small PE

memory in each of the ClearSpeed acceleration units. This reduces the number of data items that can be stored in each buffer within the double-buffering framework that has been developed. This means that each data transfer fails to reach the size required for peak efficiency(as discussed in Chapter 4). This problem is made even worse when operating in double precision and the buffer size must be reduced even further, resulting in even worse performance.

The performance difference between the manual and automatic ports on the GPU is small being less than 5% in all cases. However, there is a considerable difference between the performance of the manual and automatic ports on ClearSpeed. The reason for this is that because the N-Body simulation executes in a sequential sequence of timesteps, the output of one becoming the input of the next one. The automatic port on ClearSpeed, duplicates the input data between the two chips, and splits the output data between them. Then, at the end of a timestep, the output data are loaded back to the CPU, merged and then loaded back to both chips on the ClearSpeed card. However, the manual port, more intelligently, does not split the data, instead it loads a copy of the input and output dataset onto one chip and loads pointers to that data onto the other chip. This means that the output dataset can, instead of loading the data back to the CPU and then saving it back to the ClearSpeed card, swap memory pointers around. The performance difference between these two methods was found to be significant with the results showing that the method employed by the human developer does provide a significant performance improvement over the method employed by the automatic port.

Comparison with Existing Work

Table 6.15 shows that the N-Body application has achieved a high peak performance and a high memory bandwidth for the GPU in both single and double precision. This is an excellent result for automatically generated code. However, there are still improvements that could be made. Nyland et al[109] in 2008 developed a more efficient GPU version of the all pairs algorithm and have achieved performance of up to 163GFlop/s in single precision. Their port arranges all the bodies into tiles, with the bodies within a tile arranged in rows and columns and stored in shared memory. This method allows significant data reuse and this is what allows the increase in performance between the N-body port presented here, which is memory bound, and that in [109].

There is also evidence that an improved implementation would allow for better performance on a ClearSpeed accelerator. In their presentation at a ClearSpeed user group meeting Steinke claimed up to 3.4x improvement when comparing an N-Body simulation to a single Operon 2.8Ghz core, although very little implementation detail is provided.

Device	Peak Performance GFlop/s	Peak Memory Bandwidth MB/s	Peak Transfer Rate MB/s
Single Precision			
GPU	64.01	97.56	0.04
ClearSpeed	0.69	1.05	0.02
Double Precision			
GPU	25.31	77.31	0.04
ClearSpeed	0.47	1.43	0.05

Table 6.15: N-Body Application Peak Performance

6.2.4 Monte Carlo Methods

The name Monte Carlo refers to a technique of simulating a physical processes using a stochastic model [90] [115].

The simulation chosen for implementation is one simulating light propagation in an infinite medium with isotropic scattering. The source code used for this program is based on source provided by [30] and a flowchart describing the steps taken in the simulation is shown in Figure 6.7.

In this simulation, each photon is first launched and moved through the medium. Then a fraction of the photon's weight will be absorbed. If the remaining weight is above a minimum the direction of the photon changes and the previous steps are repeated. If the weight is below the minimum then a roulette is conducted to determine if the photon will continue or not. If the current photon does not survive then the simulation continues with the next photon.

Data Size n	Execution Time (Seconds)							
	CPU	Quad Core CPU	GPU			ClearSpeed		
			Human Port	System Port	% Diff	Human Port	System Port	% Diff
100000	1.55	0.39	1.03	1.03	0%	0.05	0.05	0%
500000	7.85	1.96	1.15	1.16	1%	0.15	0.16	7%
750000	11.56	2.89	1.23	1.24	1%	0.21	0.22	5%
1000000	15.45	3.86	1.31	1.33	2%	0.27	0.28	4%
2500000	38.45	9.61	1.78	1.81	2%	0.64	0.68	6%
5000000	76.91	19.22	2.55	2.62	3%	1.24	1.33	7%

Table 6.16: Single Precision Execution Times for Monte Carlo Application

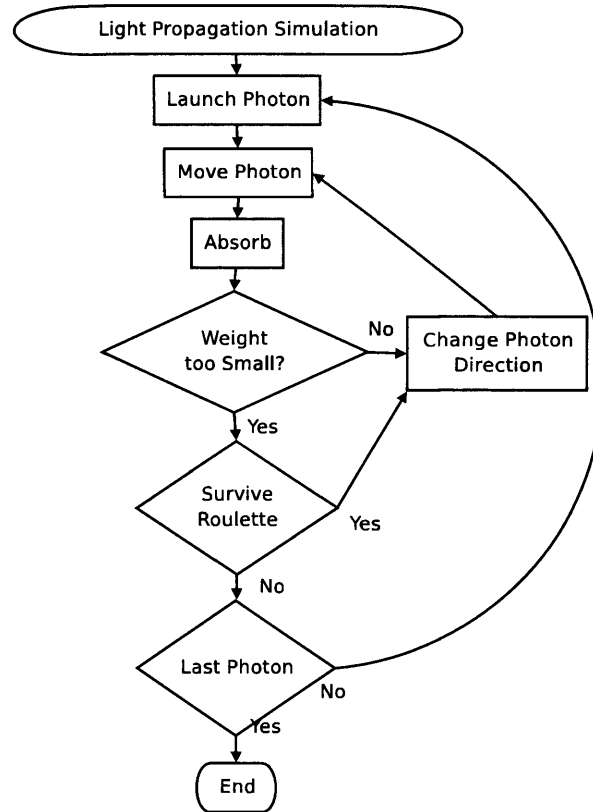


Figure 6.7: Flowchart of the Monte Carlo Technique used based on [115].

Data Size n	Execution Time (Seconds)							
	CPU	Quad Core CPU	GPU			ClearSpeed		
			Human Port	System Port	% Diff	Human Port	System Port	% Diff
100000	1.54	0.39	1.05	1.05	0%	0.04	0.05	25%
500000	7.71	1.93	1.23	1.24	1%	0.13	0.14	8%
750000	11.35	2.84	1.34	1.35	1%	0.18	0.20	11%
1000000	15.23	3.81	1.46	1.47	1%	0.23	0.25	9%
2500000	37.94	9.49	2.15	2.18	1%	0.55	0.60	9%
5000000	75.98	18.99	3.29	3.41	4%	1.08	1.18	10%

Table 6.17: Double Precision Execution Times for Monte Carlo Application

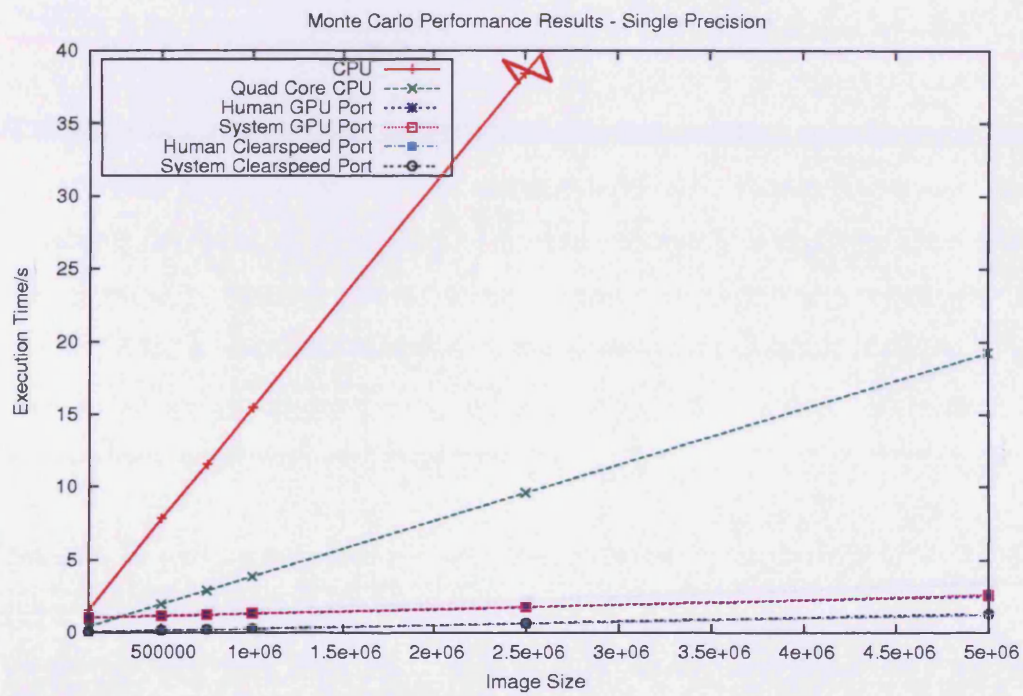


Figure 6.8: Graph of Single Precision Execution Times for Monte Carlo Application.

No. Photos	Single Precision Performance			Double Precision Performance		
	CPU	ClearSpeed	Optimal Device	CPU	ClearSpeed	Optimal Device
100000	0.90s	7.8s	C5	0.97s	7.80s	C5
500000	1.88s	12.2s	C5	1.96s	13.7s	C5
1000000	2.38s	16.1s	C5	2.47s	14.3s	C5
5000000	2.90s	19.2s	C5	2.99s	16.3s	C5
10000000	3.40s	21.6s	C5	3.49s	18.0s	C5

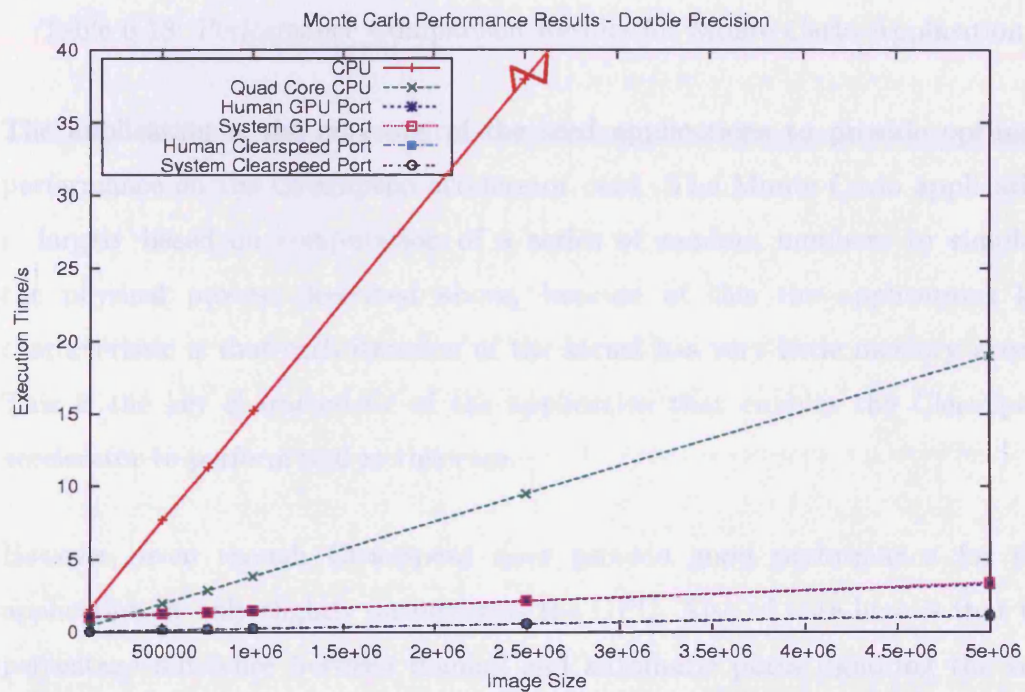


Figure 6.9: Graph of Double Precision Execution Times for Monte Carlo Application.

No Photons n	Single Precision Performance			Double Precision Performance		
	GPU	ClearSpeed	Optimal Device	GPU	ClearSpeed	Optimal Device
100000	0.90x	7.8x	CS	0.37x	7.80x	CS
500000	1.69x	12.25x	CS	1.56x	13.79x	CS
750000	2.33x	13.14x	CS	2.10x	14.20x	CS
1000000	2.90x	13.78x	CS	2.59x	15.24x	CS
2500000	5.31x	14.13x	CS	4.35x	15.82x	CS
5000000	7.34x	14.45x	CS	5.57x	16.09x	CS

Table 6.18: Performance Comparison Results for Monte Carlo Application

The application is the only one of the seed applications to provide optimum performance on the ClearSpeed accelerator card. The Monte Carlo application is largely based on computation of a series of random numbers to simulate the physical process described above, because of this the applications key characteristic is that each iteration of the kernel has very little memory access. This is the key characteristic of the application that enables the ClearSpeed accelerator to perform well in this case.

However, even though ClearSpeed does provide good performance for this application, it only slightly outperforms the GPU. Also of note here is that the percentage difference between manual and automatic ports (ignoring the very smallest case) is approximately 10%.

Finally, it is noticed that the ClearSpeed accelerator provides near identical, or sometimes better, performance in double precision but the GPU provides slower performance in double precision. This is due to the fact that the ClearSpeed accelerator only possesses double precision computation units, which perform both single and double precision calculations while the GPU has single precision floating point units and, a smaller number of double precision units.

Comparison with Existing Work

A Monte Carlo credit risk application has been one of ClearSpeed's success stories. In this example they have shown performance of ≈ 26 TFlop/s when running a 42U rack fully loaded with CATS-700 systems (giving a total of 1008 ClearSpeed chips)[37], it is also claimed that, in this example, ClearSpeed will outperform a NVIDIA Tesla by 2.7x[37]. Using the headline 26 TFlop/s figure quoted in [37] it can be calculated that each ClearSpeed board will be nearing its peak performance of 50 GFlop/s, whereas the performance of the automatic port presented in Table 6.19 only achieved 13 GFlop/s. When examining this table, it should also be noted that in this example the transfer time between the CPU and device's memories was too quick to measure.

Work has also been conducted to port Monte Carlo applications to the GPU and Alerstam et al[11] have ported a light propagation simulation, very similar to that presented here, to the GeForce 8800GT. Their results show that the GPU achieved a 1080x speed-up over an Intel Pentium 4 HT 3.4 GHz, a figure that is far superior to the ≈ 15 x speed-up that have been achieved with an automatic port.

As with many applications running on an acceleration device, this application is memory bound. The key improvement that would need to be made in order to improve performance is altering the way in which the photon data are updated, currently this must be read into the memory of the computational unit, updated and then written back to the device's memory. This would need to be changed to allow the data for each photon to be cached in a more local memory (PE Memory for ClearSpeed and shared memory for CUDA), so that computation does not stall while waiting for data from the GPU/ClearSpeed device's memory.

Device	Peak Performance GFlop/s	Peak Memory Bandwidth MB/s	Peak Transfer Rate MB/s
Single Precision			
GPU	4.45	2.21	✗
ClearSpeed	13.43	6.67	✗
Double Precision			
GPU	4.24	4.22	✗
ClearSpeed	15.41	15.32	✗

Table 6.19: Monte-Carlo Application Peak Performance

6.3 Test Applications

With the seed applications now run, and all performance data gathered so that there are no gaps in the performance database, the test applications can now be run.

Initially, the classification model that results from the data gathered from the seed applications is shown in Figure 6.10.

This section will now show how the system performs when executing three more complex applications.

6.3.1 Fast Fourier Transform

The Fast Fourier transform is based on the Discrete Fourier Transform which is shown in Equation 6.1 [39], where j is an integer ranging from 0 to $N - 1$.

$$X_j = \sum_{n=0}^{N-1} x_n e^{\frac{2\pi i}{N} jn} \quad (6.1)$$

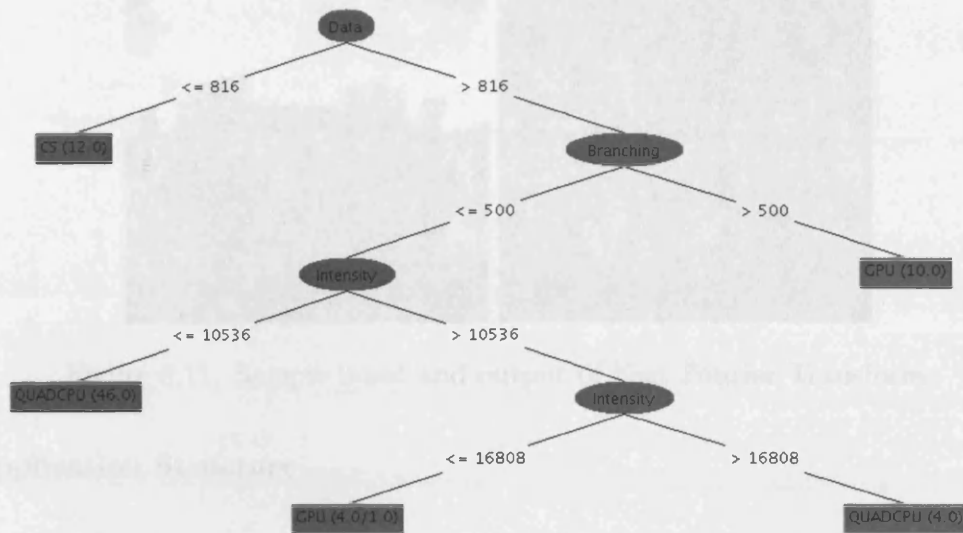


Figure 6.10: Classification Model Produced after Seed Applications.

This discrete version of the transform has a computational complexity of $O(N^2)$. So, generally speaking, is always computed by using more efficient methods known as Fast Fourier Transforms which have computational complexity of the order $O(N \log_2 N)$. The algorithm that has been chosen for this test case is the most popular of these methods: the Cooley-Tukey algorithm [39].

The actual implementation that has been created is a 2D , Radix-2 FFT. This implementation computes the FFT of an input image, a sample of which is shown in Figure 6.11. This computation is done by computing a one dimensional FFT, using the Cooley-Tukey algorithm, of each index(rows and columns) of the original input [116]. i.e, if the input image is A and the output is X then the calculation would be:

$$X = FFT - of - rows(FFT - of - columns(A))$$



Figure 6.11: Sample input and output of Fast Fourier Transform.

Application Structure

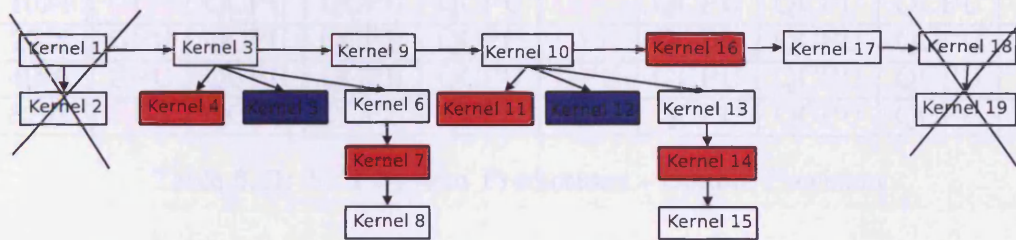


Figure 6.12: Fast Fourier Transform Application Structure

The overall structure of the generated kernel tree for the FFT application is shown in Figure 6.12. In this diagram, the kernels that are crossed out have been discarded because they contain file IO, the kernels highlighted in blue have been discarded because the loops that formed them were not deterministic and the kernels highlighted in red contain loop dependencies.

It should be noted that kernels 10, 13 and 15 are duplicates of kernels 3, 6, and 8 respectively. This is because the FFT algorithm calls for the rows of the image to be transformed, followed by the columns. Kernel 9 re-orientates the data-set in memory, in essence performing a 90° rotation on the image. Kernel 17 then scales the output from the FFT back into a format that can be output as an image file.

System Predictions

Data Size	Kernels								Optimal Device
	3	6	8	9	10	13	15	17	
512	GPU	QCPU	QCPU	QCPU	GPU	QCPU	QCPU	QCPU	GPU
1024	GPU	QCPU	QCPU	QCPU	GPU	QCPU	QCPU	QCPU	GPU
2048	GPU	QCPU	QCPU	QCPU	GPU	QCPU	QCPU	QCPU	GPU
4096	GPU	QCPU	QCPU	QCPU	GPU	QCPU	QCPU	QCPU	GPU
8192	GPU	QCPU	QCPU	QCPU	GPU	QCPU	QCPU	QCPU	GPU

Table 6.20: FFT System Predictions - Single Precision

Data Size	Kernels								Optimal Device
	3	6	8	9	10	13	15	17	
512	GPU	QCPU	QCPU	QCPU	GPU	QCPU	QCPU	QCPU	GPU
1024	GPU	QCPU	QCPU	QCPU	GPU	QCPU	QCPU	QCPU	GPU
2048	GPU	QCPU	QCPU	QCPU	GPU	QCPU	QCPU	QCPU	GPU
4096	GPU	QCPU	QCPU	QCPU	GPU	QCPU	QCPU	QCPU	GPU
8192	GPU	QCPU	QCPU	QCPU	GPU	QCPU	QCPU	QCPU	GPU

Table 6.21: FFT System Predictions - Double Precision

Tables 6.20 and 6.21 show the predictions returned by the system for the single and double precision FFT applications with differing data-set sizes. It is obvious from the tables (and will be later shown) that ClearSpeed is not competitive for this application. However, the GPU is predicted as the optimum device for all data-set sizes in both single and double precision. In these tables any items highlighted in red show where an incorrect prediction has been made (this will be discussed further in section 6.3.1) and the optimum device column shows the overall prediction that the system generates for the application as a whole (as described in Chapter 5).

It is interesting to note that the system predicts the quad-core CPU as the optimum device for all kernels apart from Kernels 3 and 10 where it predicts the GPU, and it is for this reason that the system predicts the GPU as the optimum device for the application. This decision is taken because as long as one

kernel provides a performance benefit, then the application will give increased performance executing on a system with a GPU, even if only one of the kernels actually executes on the GPU.

Validating the Predictions

With the system's predictions now made, they must be verified. To do this the execution times, for each individual kernel are shown (details on how these data are gathered are described in Chapter 5). Tables 6.22 and 6.23 show the relative execution times of the GPU and ClearSpeed devices compared to the quad-core CPU. In these tables, a positive value indicates that the kernel took longer to execute on the device than on the quad-core CPU, while a negative value shows that a kernel executed faster on the acceleration device than on the quad-core CPU. The decision to present measurements in this manner, was taken because this section is focused on the relative performance of the kernels and not on the amount of acceleration achieved. Information on the overall performance of the application and the acceleration achieved is presented in Section 6.3.1.

Data Size	Kernel Execution Time Relative to Quad-Core CPU in Seconds									
	3		6		8		9		17	
	CS	GPU	CS	GPU	CS	GPU	CS	GPU	CS	GPU
512	0.41	0.96	0.03	0.01	✗	✗	0.06	0.96	0.04	0.96
1024	1.76	0.56	✗	0.01	✗	✗	0.18	0.95	0.14	0.96
2048	7.27	0.99	✗	0.04	✗	✗	0.17	0.92	0.57	1.00
4096	48.76	1.13	✗	0.13	✗	✗	3.29	0.79	2.14	1.11
8192	206.04	3.42	✗	✗	✗	✗	✗	0.35	8.01	1.58

Table 6.22: Kernel Execution Times for FFT Application - Single Precision

From these data, the incorrect predictions are highlighted in red in Tables 6.20 and 6.21. This shows that the system made a total of 20 false predictions out of 80. That is a success rate of 75%. The errors that have been made are relating

Data Size	Kernel Execution Time Relative to Quad-Core CPU in Seconds									
	3		6		8		9		17	
	CS	GPU	CS	GPU	CS	GPU	CS	GPU	CS	GPU
512	0.26	0.96	0.04	0.01	✗	✗	0.05	0.96	0.04	0.96
1024	1.41	0.97	✗	0.01	✗	✗	0.15	0.96	0.13	8.96
2048	5.69	1.02	✗	0.04	✗	✗	0.62	0.95	0.44	1.01
4096	38.98	1.64	✗	0.13	✗	✗	2.12	0.67	1.48	1.15
8192	✗	9.08	✗	✗	✗	✗	✗	1.09	✗	1.74

Table 6.23: Kernel Execution Times for FFT Application - Double Precision

to kernels 3 and 10. Both of these kernels should be executing on the quad core CPU for all data-sets, meaning that this application should be predicted to execute optimally on the quad-core CPU.

The reason that the system performs poorly in regards to kernels 3 and 10 is that these kernels are all mathematically intense kernels, but they execute relatively few times (each kernel only executes once per row). This type of kernel has not been seen before by the system, as all the seed kernels it encountered were kernels with high execution counts.

However, even though the system has not achieved complete accuracy with its initial predictions for this application, it is able to self-modify its own classification model, by using the performance data that has been gathered. The resultant classification model is shown in Figure 6.13 and it can be seen that the model has changed based on the new performance data. This new model will be used to make the predictions for the next test application: The Canny Edge Detector.

Overall Application Performance

Finally, the overall performance of the FFT application should be examined. Tables 6.24, 6.25 and 6.26 and Figures 6.14 and 6.15 show the overall performance

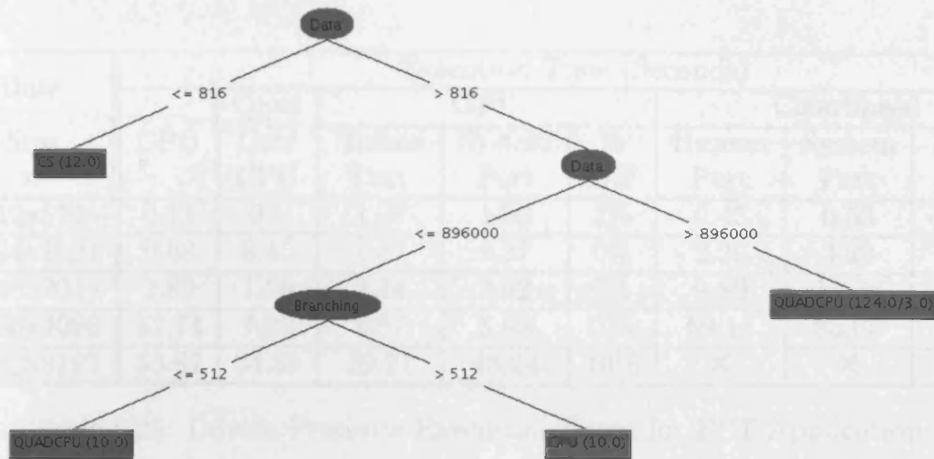


Figure 6.13: Classification Model Produced after FFT Application has been profiled.

data for the application, executing with the optimal kernel configuration for each data-set on each device.

Data Size n	Execution Time (Seconds)							
	CPU	Quad Core CPU	GPU			ClearSpeed		
			Human Port	System Port	% Diff	Human Port	System Port	% Diff
512x512	0.12	0.08	1.07	1.08	0%	0.82	0.88	7%
1024x1024	0.63	0.42	1.29	1.35	5%	3.5	3.94	12%
2048x2048	2.75	1.80	2.22	2.3	4%	22.08	23.21	5%
4096x4096	11.64	7.59	6.13	6.51	6%	133.89	137.98	3%
8192x8192	50.15	32.28	25.8	27.7	7%	397.09	444.18	12%

Table 6.24: Single Precision Execution Times for FFT Application

Data Size n	Execution Time (Seconds)							
	CPU	Quad Core CPU	GPU			ClearSpeed		
			Human Port	System Port	% Diff	Human Port	System Port	% Diff
512x512	0.14	0.1	1.07	1.09	2%	0.45	0.63	40.00%
1024x1024	0.68	0.45	1.37	1.37	0%	2.29	3.28	43%
2048x2048	2.89	1.86	2.44	2.62	7%	9.89	13.36	35%
4096x4096	12.74	7.38	7.87	8.68	10%	69.11	85.65	24%
8192x8192	53.87	34.59	39.21	43.24	10%	×	×	×

Table 6.25: Double Precision Execution Times for FFT Application

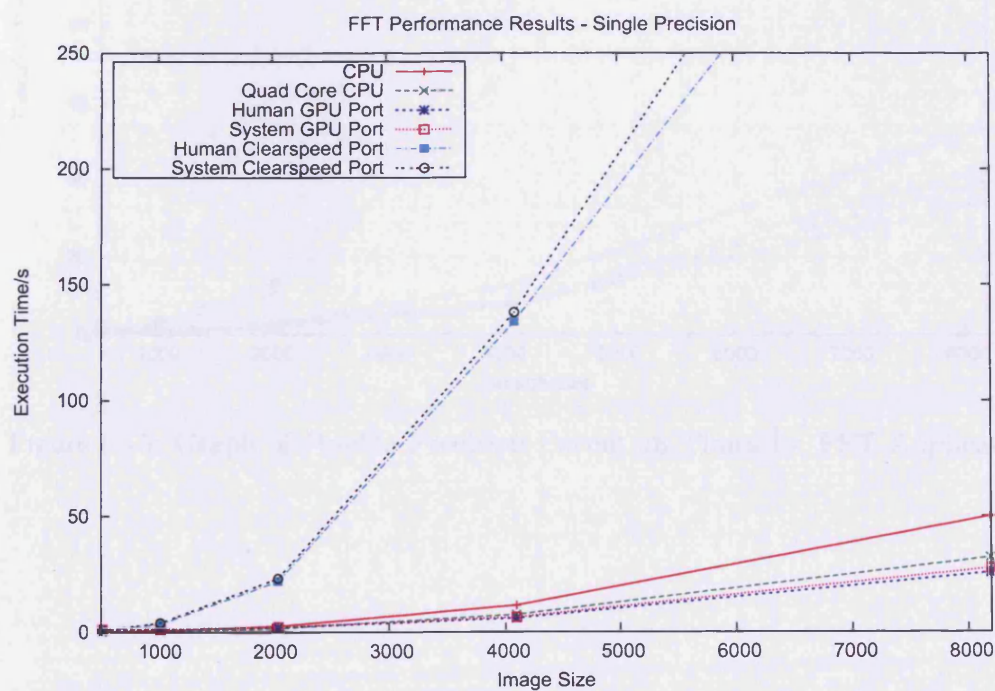


Figure 6.14: Graph of Single Precision Execution Times for FFT Application.

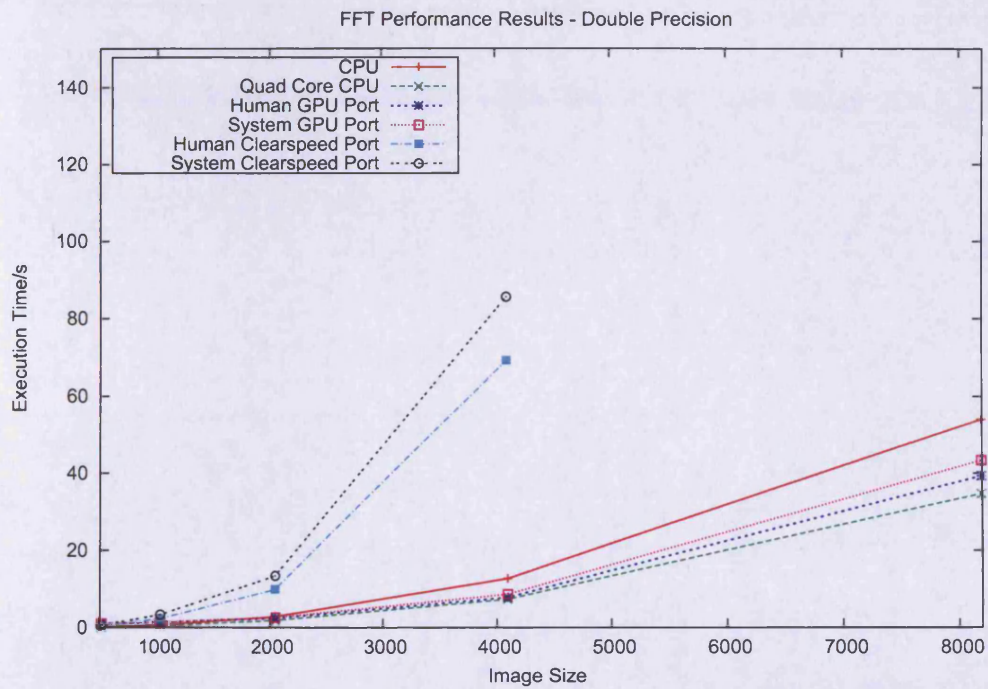


Figure 6.15: Graph of Double Precision Execution Times for FFT Application.

Data Size	Single Precision Performance			Double Precision Performance		
	GPU	ClearSpeed	Optimal Device	GPU	ClearSpeed	Optimal Device
512x512	0.07x	0.09x	QCPU	0.09x	0.16x	QCPU
1024x1024	0.31x	0.11x	QCPU	0.33x	0.14x	QCPU
2048x2048	0.78x	0.08x	QCPU	0.71x	0.14x	QCPU
4096x4096	1.17x	0.06x	GPU	0.85x	0.09x	QCPU
8192x8192	1.17x	0.07x	GPU	0.80x	✗	QCPU

Table 6.26: Performance Comparison Results for FFT Application

This data confirms that the ClearSpeed device is completely uncompetitive for this application. This is due to the fact that the FFT algorithm that is used modifies the data in-place. This means that we cannot split the data-set between the two memory chips on the ClearSpeed card, meaning that we must incur the overhead of one chip accessing another chip's memory. Secondly, the FFT algorithm does not access memory in a regular pattern, this means that double-buffering, which is often used to hide the memory latency, cannot be used. More interestingly, ClearSpeed actually performs better in the double precision case. This is because ClearSpeed can achieve faster memory transfer rates with larger transfers (as described in Section 4.4.5), this means that transfers of double precision data are naturally more efficient. Secondly, ClearSpeed only contains a 64bit floating point unit, meaning that at single precision only half of each floating point unit is being utilised. Finally, due to the limitations on ClearSpeed's device memory, the largest test case in double precision was unable to be executed. It should be noted at this point (and will be discussed later) that ClearSpeed provides a sample implementation of the FFT application, which was not used in this test. This version of the FFT uses a re-factored algorithm that provides vastly improved performance, to achieve this it is assumed (due to the closed source nature of this application) that ClearSpeed's implementation leverages on the Swizzle operation [73].

The results achieved when executing the application on the GPU are interesting in that in single precision, contrary to the performance measured when examining each individual kernel, the GPU does outperform the quad-core CPU for the largest two data-sets. The reason for this is because the FFT application consists of multiple kernels, asynchronous transfer of data between the host's memory and the GPU's memory can take place. This means that in essence data can be streamed back to the CPU's memory before computation has been completed and by extension, data transfers for the execution of the next kernel can begin sooner.

When examining the performance differences between the automatically generated code and hand-ported code, we can see that there is very little difference between them on the GPU. However, there is a significant difference in performance when considering ClearSpeed. The reason for this, is the manual port on ClearSpeed is able to reduce memory transfers back and forward to the host, by combining some of the kernels. This means that the entire FFT application is combined into one larger kernel, with an integer variable acting as a mode flag, determining what kernel is actually executed.

Comparison with Existing Work

The FFT is a core part of many computing applications and as such it is provided as a library on both ClearSpeed and CUDA[73][100]. The CUFFT library when running a 1D radix-2 FFT gives performance of up to $\approx 350\text{GFlop/s}$ in single precision and $\approx 100\text{GFlop/s}$ [107] in double precision. It should be noted however, that currently performance figures are only available for NVIDIA's top of the range FERMI card, which is significantly more powerful than the GPU used in this thesis. In comparison, ClearSpeed's implementation provides performance of

up to 19.5 GFlop/s for a 1D FFT and 16.2 GFlop/s for a 2D FFT [38].

Another FFT implementation is presented by Govindaraju et al[59], in this paper the authors utilise the Stockham formulation of the FFT, commenting that the Cooley-Tukey can be expensive due to incoherent memory accesses[59]. As part of this work the authors implement several FFT algorithms and then compare the performance using a NVIDIA GTX280 (roughly equivalent in single precision to the TESLA card used in this thesis). The highest performance achieved was approaching 300 GFlop/s, providing a 2-4x improvement over CUFFT running on the same GPU[59]. One key point that is made in the paper, is that depending on: the radix of the FFT, the problem size and the architecture, the correct choice of FFT algorithm is important to get maximum performance. This shows the existence of a complex decision space and the ideal circumstances for when a decision making system is needed, as when selecting the optimal device for a FFT application, the selection of the algorithm, and by extension the device, both depend not only on the size but also on the characteristics of the input data-set.

When comparing the results achieved in literature to those presented in Table 6.27 it can be seen that the automatic port is far from achieving peak performance. Probably the biggest fault here is the fact that the Cooley-Tukey algorithm is far from optimal for the GPU/ClearSpeed architecture, being completely memory bound due to frequent incoherent memory accesses[59].

6.3.2 Canny Edge Detector

The next application that was trialled using the system is a classic image processing algorithm: The Canny Edge detector [123] [28].

This method of edge detection is a multi step algorithm which involves the

Device	Peak Performance GFlop/s			Peak Memory Bandwidth MB/S			Peak Transfer Rate MB/S		
	3	9	17	3	9	17	3	9	17
Single Precision									
GPU	2.21	0.52	1.74	2.14	0.75	0.35	1.95	2.68	2.67
ClearSpeed	0.07	0.08	0.39	0.07	0.12	0.08	0.7	0.75	0.72
Double Precision									
GPU	1.22	0.41	2.08	2.36	1.19	0.83	1.97	3.48	2.67
ClearSpeed	0.09	0.13	1.64	0.17	0.37	0.65	0.68	0.77	0.89

Table 6.27: FFT Application Peak Performance

following steps [123] [28]:

1. Noise Reduction: The noise reduction step involves blurring the input image by convolving it with a Gaussian Filter. The one that is used in this implementation is shown in Table 6.28.

2. Detection: The detection step that is used in our implementation of the Canny Edge detector is the Sobel. Firstly the image will be convolved using two masks, A and B , which are shown in Table 6.3. Then the magnitude of gradient of each pixel is then calculated using:

$$|G| = \sqrt{A^2 + B^2}$$

Additionally the angle of the gradient θ is then is computed using:

$$\theta = \arctan\left(\frac{A}{B}\right)$$

Finally, θ' is computed by rounding θ to one of four directions 0° , 45° , 90° or 135°

3. Non-Maximum Suppression: The Non-Maximum Suppression step will

ensure that the edges that have been found by the previous step are 1 pixel wide, by checking that, on an edge, only the pixels with the highest gradient magnitude are kept. This is computed by examining neighbouring pixels according to the angle of the gradient θ' which was computed previously and if the current pixel being examined is greater than its neighbours it is kept as an edge pixel, otherwise it is discarded.

4. Thresholding by Hysteresis: The final step in the algorithm is hysteresis thresholding. This step is performed by taking two thresholds T_{low} and T_{high} . Pixels that fall above T_{high} are retained, while pixels falling below T_{low} are discarded, by pixels that fall between the two thresholds are kept only if they form an edge with the pixels that fall above T_{high} .

$$\frac{1}{159} * \begin{array}{|c|c|c|c|c|} \hline 2 & 4 & 5 & 4 & 2 \\ \hline 4 & 9 & 12 & 9 & 4 \\ \hline 5 & 12 & 15 & 12 & 5 \\ \hline 4 & 9 & 12 & 9 & 4 \\ \hline 2 & 4 & 5 & 4 & 2 \\ \hline \end{array}$$

Table 6.28: Gaussian Filter Used for Blurring

Sample input and sample output data for the implemented Canny Edge detector is shown in Figure 6.16.

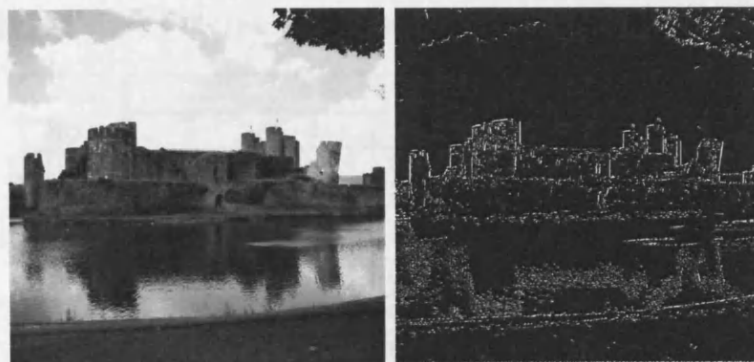


Figure 6.16: Sample input and output of a Canny Edge Detector .

Application Structure

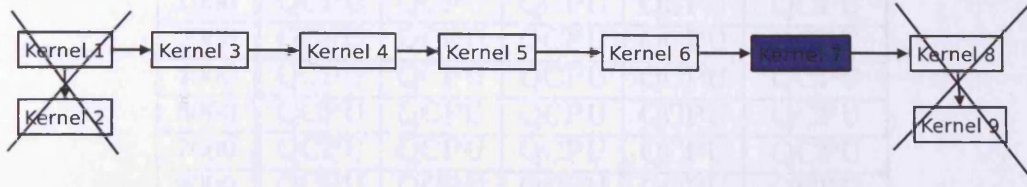


Figure 6.17: Canny Application Structure

The overall structure of the generated kernel tree for the Canny application is shown in Figure 6.17. In this diagram, the kernels that are crossed out have been discarded because they contain file IO and the kernels highlighted in blue have been discarded because they contained recursion. In the application structure kernel 3 is the kernel for the Gaussian blur, kernel 4 is the kernel for the Sobel edge detection and the computation of the angle of gradient of the edges, kernel 5 is non-maximal suppression and kernel 6 is the initialisation of the output array for the hysteresis(Kernel 7) to zero.

System Predictions

The predictions made by the system using the classification model shown earlier (Figure 6.13) are shown in Tables 6.29 and 6.30.

Data Size	Kernels				Optimal Device
	3	4	5	6	
1000	QCPU	QCPU	QCPU	QCPU	QCPU
2000	QCPU	QCPU	QCPU	QCPU	QCPU
4000	QCPU	QCPU	QCPU	QCPU	QCPU
6000	QCPU	QCPU	QCPU	QCPU	QCPU
7000	QCPU	QCPU	QCPU	QCPU	QCPU
8000	QCPU	QCPU	QCPU	QCPU	QCPU

Table 6.29: Canny System Predictions - Single Precision

Data Size	Kernels				Optimal
	3	4	5	6	Device
1000	QCPU	QCPU	QCPU	QCPU	QCPU
2000	QCPU	QCPU	QCPU	QCPU	QCPU
4000	QCPU	QCPU	QCPU	QCPU	QCPU
6000	QCPU	QCPU	QCPU	QCPU	QCPU
7000	QCPU	QCPU	QCPU	QCPU	QCPU
8000	QCPU	QCPU	QCPU	QCPU	QCPU

Table 6.30: Canny System Predictions - Double Precision

Validating the Predictions

In order to validate the predictions that have been made the application was analysed on a per-kernel basis. These data is shown in Tables 6.31 and 6.32. From this we can see that the system correctly predicts all of the kernels and that it predicts that this application will perform optimally on the quad-core CPU.

Data Size	Kernel Execution Time Relative to Quad-Core CPU in Seconds							
	3		4		5		6	
	CS	GPU	CS	GPU	CS	GPU	CS	GPU
1000	0.08	0.96	0.14	0.94	0.07	0.97	0.03	0.96
2000	0.23	0.96	0.47	0.91	0.16	1.00	0.06	0.99
4000	0.86	0.99	1.77	0.74	0.60	1.11	0.28	1.11
6000	1.95	1.03	4.00	0.47	1.29	1.33	0.50	1.31
7000	2.61	1.06	5.45	0.36	1.74	1.48	0.66	1.48
8000	2.96	1.12	6.63	0.14	2.08	1.61	0.83	1.59

Table 6.31: Kernel Execution Times for Canny Application - Single Precision

This application achieved a 100% success rate with its predictions, because of this the generated decision tree should not change, however, a new tree can still be generated based on the results from the Canny application being added to the training set. This new model, taking into account all the seed applications, the FFT application and the Canny edge detector is shown in Figure 6.18. As

Data Size	Kernel Execution Time Relative to Quad-Core CPU in Seconds							
	3		4		5		6	
	CS	GPU	CS	GPU	CS	GPU	CS	GPU
1000	0.12	0.96	0.15	0.95	0.09	0.97	0.04	0.97
2000	0.44	0.98	0.58	0.92	0.31	1.02	0.32	1.00
4000	1.29	1.07	2.15	0.79	0.96	1.21	0.37	1.13
6000	2.88	1.23	4.89	0.58	1.97	1.55	0.80	1.35
7000	×	1.32	×	0.51	×	1.77	×	1.49
8000	×	2.35	×	0.30	×	1.97	×	1.66

Table 6.32: Kernel Execution Times for Canny Application - Double Precision

expected the tree has not changed from that in Figure 6.13.

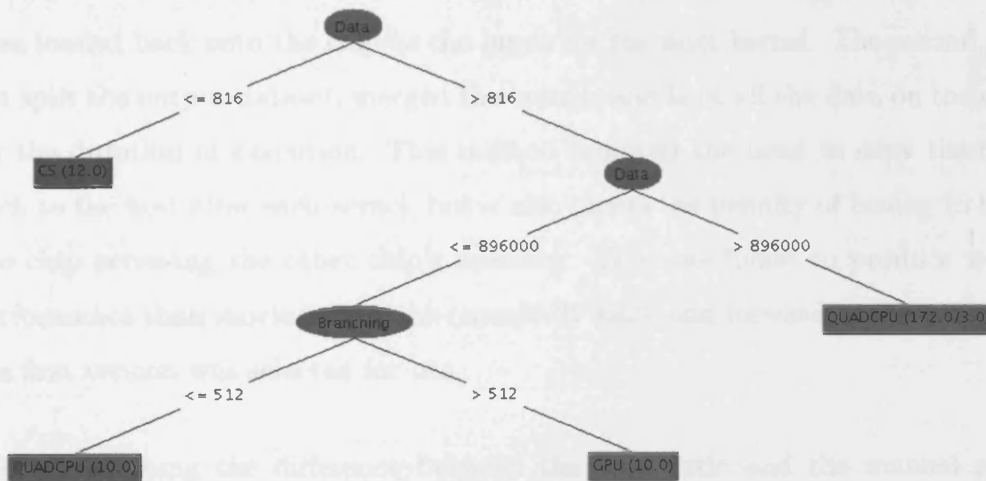


Figure 6.18: Classification Model Produced after Canny Application has been profiled.

Overall Application Performance

When analysing the overall application performance of the Canny, as shown in Tables 6.33, 6.34 and 6.35 and Figures 6.19 and 6.20 we can see that as predicted the quad-core CPU provides the best performance, although the

difference between the quad-core CPU and the automatic port is relatively small. Additionally it can be seen that the manual GPU port does actually outperform the quad-core CPU by a small margin once the data-set size reaches 6000x6000 in single precision and 7000x7000 in double precision.

ClearSpeed proves to be uncompetitive here and due to its memory arrangement, it cannot execute the higher dataset sizes. In this example, when producing the manual ClearSpeed port, two different versions were considered. The first is the version that is present here, this does not attempt to merge kernels together and instead accepts that the output of each kernel, which is split so each chip produces half the output, must be brought back to the CPU merged together and then loaded back onto the chip as the input for the next kernel. The second, did not split the output dataset, merged the kernels and kept all the data on the card for the duration of execution. This method removed the need to copy the data back to the host after each kernel, but it also incurs the penalty of having to have one chip accessing the other chip's memory. This was found to produce worse performance than moving the data repeatedly back and forward to the CPU. So the first version was selected for use.

When examining the difference between the automatic and the manual port, this example presents a considerable difference in some cases. The reason for this is that both the Gaussian Blur and Sobel kernels are able to be made more efficient by making use of the multiprocessors shared memory to store values that are used in multiple threads. Additionally, the need to rely on the concurrent loading/executing has been removed in the manual port as, in this application, we can keep the entire input and output data-sets on the GPU's memory for the duration of the execution of the kernels, without the need for it to be loaded back to the host.

Data Size n	Execution Time (Seconds)							
	CPU	Quad Core CPU	GPU			ClearSpeed		
			Human Port	System Port	% Diff	Human Port	System Port	% Diff
1000x1000	0.29	0.16	1.12	1.14	2%	0.36	0.42	17%
2000x2000	0.99	0.47	1.3	1.4	8%	1.16	1.43	23%
4000x4000	3.7	1.5	1.99	2.43	22%	4.14	5.34	29%
6000x6000	8.18	3.33	3.14	4.12	31%	9.12	11.95	31%
7000x7000	11.1	4.7	3.82	5.35	40%	11.1	16.08	45%
8000x8000	14.28	5.65	4.86	6.44	33%	15.96	19.46	22%

Table 6.33: Single Precision Execution Times for Canny Application

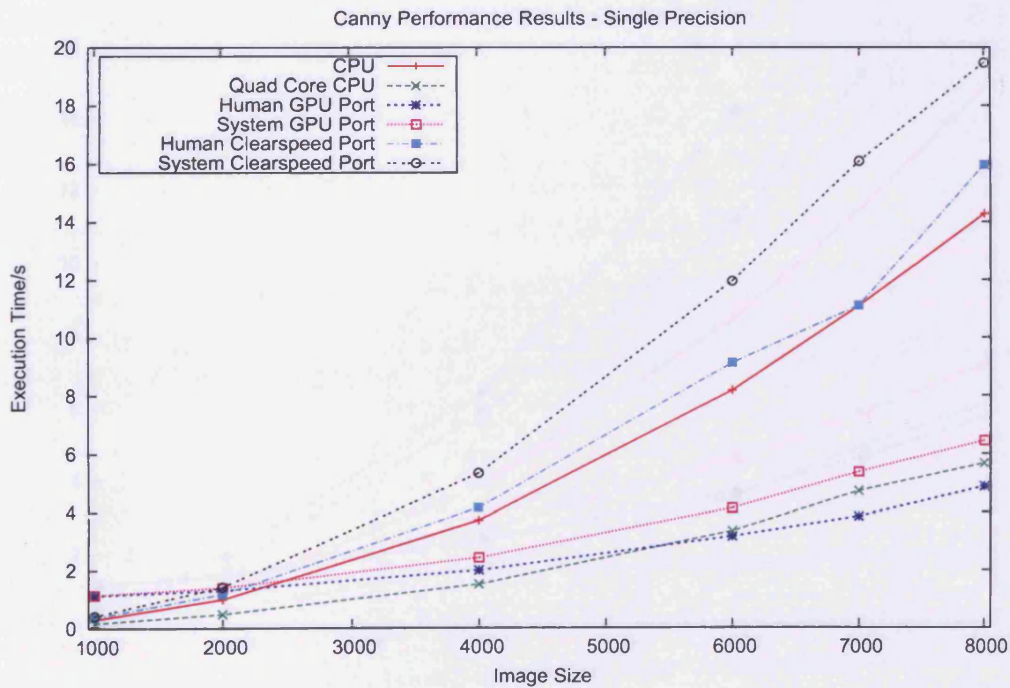


Figure 6.19: Graph of Single Precision Execution Times for Canny Application.

Data Size n	Execution Time (Seconds)							
	CPU	Quad Core CPU	GPU			ClearSpeed		
			Human Port	System Port	% Diff	Human Port	System Port	% Diff
1000x1000	0.29	0.16	1.11	1.14	3%	0.44	0.5	14%
2000x2000	1.01	0.48	1.36	1.46	7%	1.56	1.91	22%
4000x4000	3.86	1.6	2.36	2.64	12%	5.82	6.39	10%
6000x6000	8.47	3.52	3.64	4.54	25%	11.15	14.17	27%
7000x7000	11.5	4.91	4.58	5.81	27%	15.14	×	×
8000x8000	14.89	6.06	5.72	7.18	26%	×	×	×

Table 6.34: Double Precision Execution Times for Canny Application

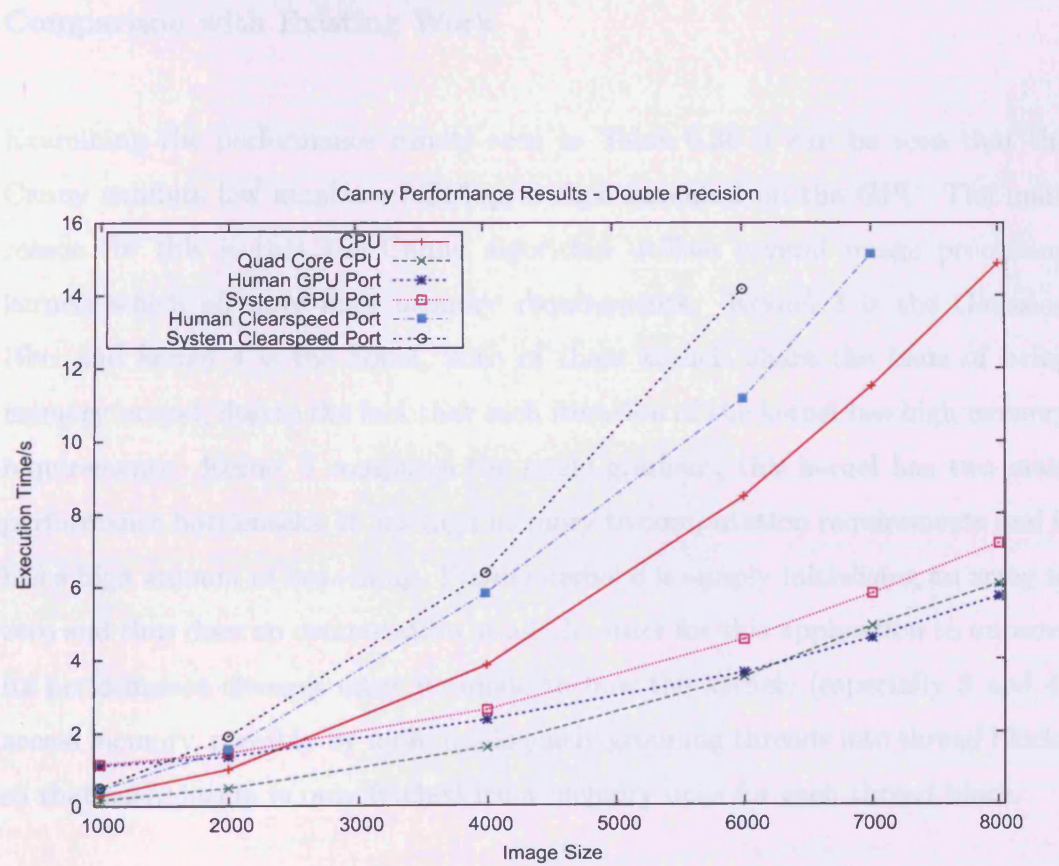


Figure 6.20: Graph of Double Precision Execution Times for Canny Application.

Data Size n	Single Precision Performance			Double Precision Performance		
	GPU	ClearSpeed	Optimal Device	GPU	ClearSpeed	Optimal Device
1000x1000	0.14x	0.38x	QCPU	0.14x	0.32x	QCPU
2000x2000	0.36x	0.33x	QCPU	0.33x	0.25x	QCPU
4000x4000	0.62x	0.28x	QCPU	0.61x	0.25x	QCPU
6000x6000	0.81x	0.28x	QCPU	0.78x	0.24x	QCPU
7000x7000	0.88x	0.29x	QCPU	0.85x	✘	QCPU
8000x8000	0.88x	0.29x	QCPU	0.84x	✘	QCPU

Table 6.35: Performance Comparison Results for Canny Application

Comparison with Existing Work

Examining the performance results seen in Table 6.36 it can be seen that the Canny exhibits low number of GFlop/s when executed on the GPU. The main reason for this is that the Canny algorithm utilises several image processing kernels which all have high memory requirements. Kernel 3 is the Gaussian Blur and kernel 4 is the Sobel, both of these kernels share the issue of being memory bound, due to the fact that each iteration of the kernel has high memory requirements. Kernel 5 computes the angle gradient, this kernel has two main performance bottlenecks; it has high memory to computation requirements and it has a high amount of branching. Finally kernel 6 is simply initialising an array to zero and thus does no computation at all. In order for this application to improve its performance changes must be made to how the kernels (especially 3 and 4) access memory, possibly by more intelligently grouping threads into thread blocks so that each datum is only fetched from memory once for each thread block.

Device	Peak Performance GFlop/s				Peak Memory Bandwidth MB/S				Peak Transfer Rate MB/S			
	3	4	5	6	3	4	5	6	3	4	5	6
Single Precision												
GPU	3.91	3.18	2.07	0.04	3.81	2.72	4.38	0.18	1.36	1.36	0.76	1.28
CS	3.36	0.78	3.73	0.07	3.27	0.67	7.89	0.29	0.17	0.08	0.15	0.16
Double Precision												
GPU	2.38	2.89	1.92	0.04	4.63	4.96	8.12	0.36	1.81	1.35	0.88	1.81
CS	0.54	0.78	0.97	0.04	1.05	1.33	4.11	0.35	0.2	0.05	0.19	0.2

Table 6.36: Canny Application Peak Performance

6.3.3 Iterative Ray Tracer

The final test application that has been selected in an iterative ray tracing application. This application will render a scene consisting of three spheres in a variety of image sizes. For each rendered image, the sizes of the spheres will be adjusted to ensure that the amount of work done in rendering each image size is the same. An example of the rendered output of the ray tracer is shown in Figure 6.21.



Figure 6.21: Sample output of Iterative Ray Tracer

The standard ray tracing algorithm (shown in Listing 6.1) is taken from [127].

Currently the ray tracer supports diffuse and specular lighting using Phong Illumination.

Listing 6.1: Ray Tracing Algorithm [127]

```
for each pixel
    shoot a ray into the image from the centre of the
      pixel
    for each step the ray takes through the image
      compute intersection with objects
      for each light source
        compute if current pixel is in
          shadow
        if not in shadow
          compute new RGB value
            from diffuse and
            specular lighting
            components
        end if
      end for
    end for
end for
```

Application Structure

The structure of the application is shown in Figure 6.22. In this application the only viable kernel is kernel 1 which executes per pixel. Kernel 2, which moves the ray through the image, is non deterministic and kernel 3, which loops through each light source, contains loop dependencies. Kernel 4 and 5 are disqualified because they contain file IO.

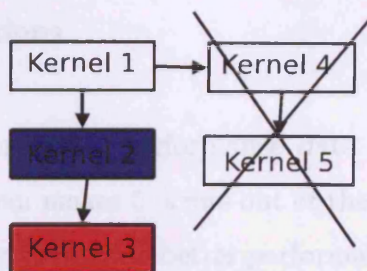


Figure 6.22: Ray Tracer Application Structure

System Predictions

The predictions that the system has made regarding this application are shown in Tables 6.37 and 6.38.

Data Size	Kernel 1	Optimal Device
500	QCPU	QCPU
1000	QCPU	QCPU
2500	QCPU	QCPU
5000	QCPU	QCPU
7500	QCPU	QCPU
10000	QCPU	QCPU

Table 6.37: Ray Tracer System Predictions - Single Precision

Data Size	Kernel 1	Optimal Device
500	QCPU	QCPU
1000	QCPU	QCPU
2500	QCPU	QCPU
5000	QCPU	QCPU
7500	QCPU	QCPU
10000	QCPU	QCPU

Table 6.38: Ray Tracer System Predictions - Double Precision

Validating the Predictions

Examining the actual per kernel performance data shown in Tables 6.39 and 6.40 reveals that the system makes 5 errors out of the 10 predictions, mistakenly predicting this application to provide better performance on the quad-core CPU.

	Kernel Execution Time Relative to Quad-Core CPU in Seconds	
Data Size	Kernel 1	
	CS	GPU
500	0.60	0.94
1000	2.45	0.90
2500	15.91	0.62
5000	78.24	-0.35
7500	165.14	-1.89
10000	✘	-2.98

Table 6.39: Kernel Execution Times for Ray Tracer Application - Single Precision

	Kernel Execution Time Relative to Quad-Core CPU in Seconds	
Data Size	Kernel 1	
	CS	GPU
500	0.61	0.95
1000	2.52	0.92
2500	16.42	0.69
5000	80.39	0.01
7500	✘	-1.38
10000	✘	-2.08

Table 6.40: Kernel Execution Times for Ray Tracer Application - Double Precision

Based on the new performance data gathered a new classification model (shown in Figure 6.23) can be built. From examining this model we can see that, because the predictions made here were poor, the model has changed significantly becoming

significantly more complex, this occurred because the dominance of the quad-core CPU in the previous example has led to the decision tree becoming overly simplified.

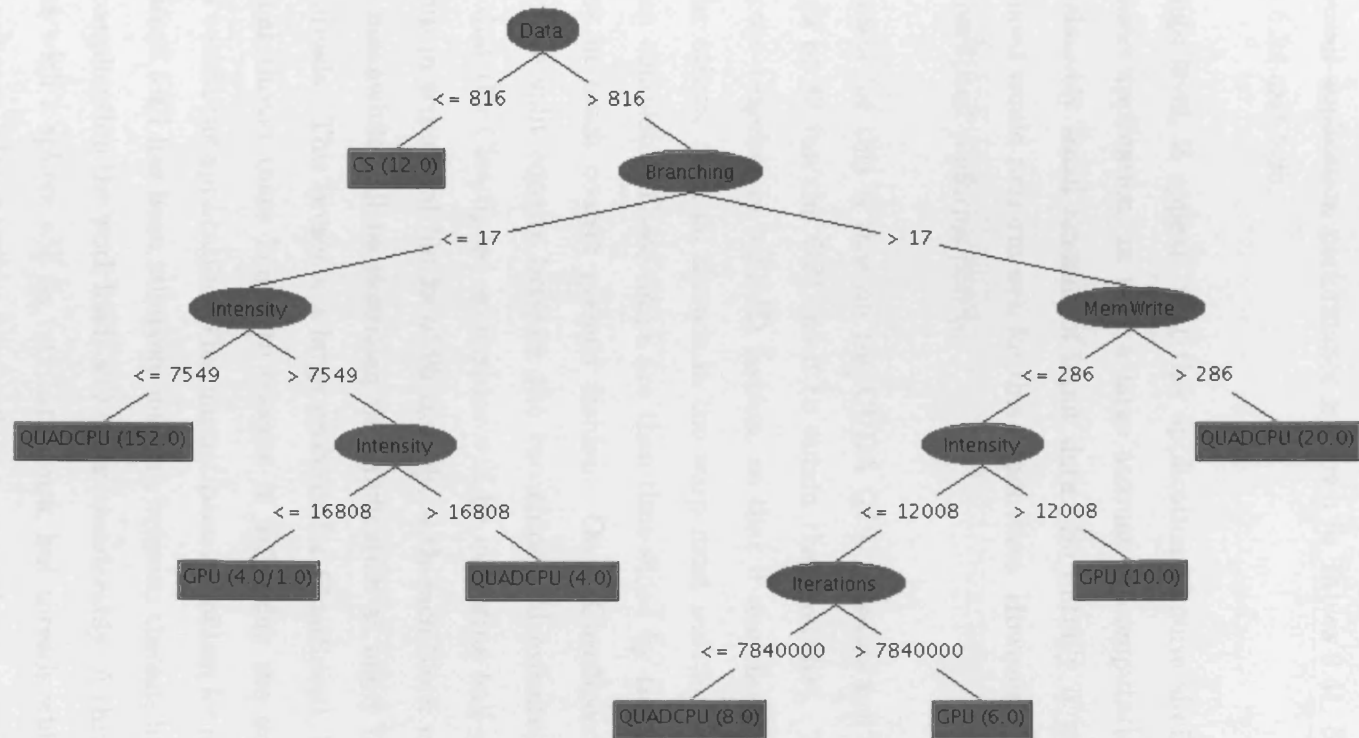


Figure 6.23: Final Classification Model

Overall Application Performance

The overall application performance is shown in Tables 6.41, 6.42 and 6.43 and graphs 6.24 and 6.25.

At a high level, it appears that this application is quite similar to the Monte Carlo seed application, in that a large amount of computation is carried out for a relatively small amount of input data. So initially it was expected that ClearSpeed would perform well for this application. However, this is not the case and ClearSpeed performs poorly.

The reason for this is that on the CUDA GPU, threads are batched together in warps of 32 (on the GPU used to obtain these results). These 32 threads all execute together in a SIMD fashion, so that if one thread does more work than the others, then all threads in the warp must wait for it to finish, before they can complete. These warps are then time-sliced by the GPU. ClearSpeed operates in much coarser grained fashion. On a ClearSpeed accelerator the threads are split equally between the two chips and executed in SIMD. This means that the ClearSpeed accelerator will be executing half of the threads on each chip in sequential blocks of 96 threads, with each block running in SIMD. CUDA meanwhile will be executing the application as many time-sliced blocks of 32 threads. This presents a large problem for ClearSpeed, because when an individual thread takes longer to execute it will delay the execution of other threads within the application. This has not been a problem for other applications as the work load has been relatively uniform between threads but within the ray tracing application the work loads will differ considerably. A thread that does not intersect with a sphere will do very little work, but threads which intersect with several spheres, plus possibly have reflections to calculate, will be many times slower due as multiple interactions between the ray and objects in the scene must

be calculated.

In order to mitigate against this two arrangements of the threads on ClearSpeed's processing elements were tried: in memory order, and grouped by locality. It was found, however, that when comparing these arrangements neither were seen to give any noticeable performance improvement. To this end we must assume, as long as we respect the limitation of not re-factoring the algorithm, that this application simply is not suitable for ClearSpeed's architecture in its current form.

Data Size n	Execution Time (Seconds)							
	CPU	Quad Core CPU	GPU			ClearSpeed		
			Human Port	System Port	% Diff	Human Port	System Port	% Diff
500x500	0.23	0.17	1.09	1.11	3%	0.69	0.74	7%
1000x1000	0.43	0.18	1.14	1.16	2%	2.7	3.14	16%
2500x2500	2.64	1.08	1.68	1.74	4%	17.36	17.5	1%
5000x5000	10.27	4.19	3.54	3.84	9%	83.2	84.51	2%
7500x7500	22.28	8.99	6.47	7.31	13%	176.57	179.03	1%
10000x10000	35.43	15.59	10.92	12.15	11%	✗	✗	✗

Table 6.41: Single Precision Execution Times for Iterative Ray Tracer

Data Size n	Execution Time (Seconds)							
	CPU	Quad Core CPU	GPU			ClearSpeed		
			Human Port	System Port	% Diff	Human Port	System Port	% Diff
500x500	0.23	0.16	1.1	1.08	2%	0.74	0.74	0%
1000x1000	0.47	0.20	1.18	1.18	0%	2.78	2.82	2%
2500x2500	2.84	1.11	1.81	1.9	5%	17.47	18.1	4%
5000x5000	10.99	4.22	4.16	4.39	6%	85.69	88.04	3%
7500x7500	24.07	9.28	7.72	8.44	9%	✗	✗	✗
10000x10000	36.77	14.80	13.26	13.92	5%	✗	✗	✗

Table 6.42: Double Precision Execution Times for Iterative Ray Tracer

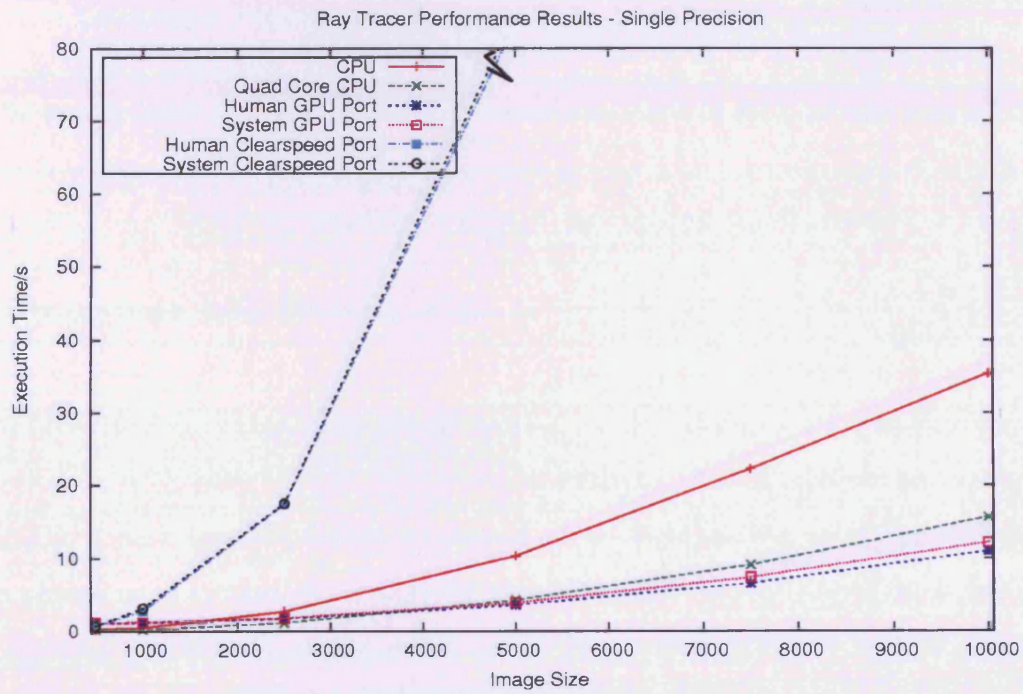


Figure 6.24: Graph of Single Precision Execution Times for Iterative Ray Tracer.

Data Size	Single Precision Performance			Double Precision Performance		
	CPU	CleartSpeed	Optima Device	GPU	CleartSpeed	Optima Device
1000	0.13s	0.23s	QCPU	0.12s	0.22s	QCPU
10000	0.16s	0.33s	QCPU	0.15s	0.27s	QCPU
20000	0.62s	0.65s	QCPU	0.58s	0.62s	QCPU
50000	1.90s	0.65s	GPU	0.90s	0.65s	GPU
75000	1.28s	0.64s	GPU	1.28s	0.64s	GPU
100000	1.25s	X	GPU	1.6s	2.3s	GPU

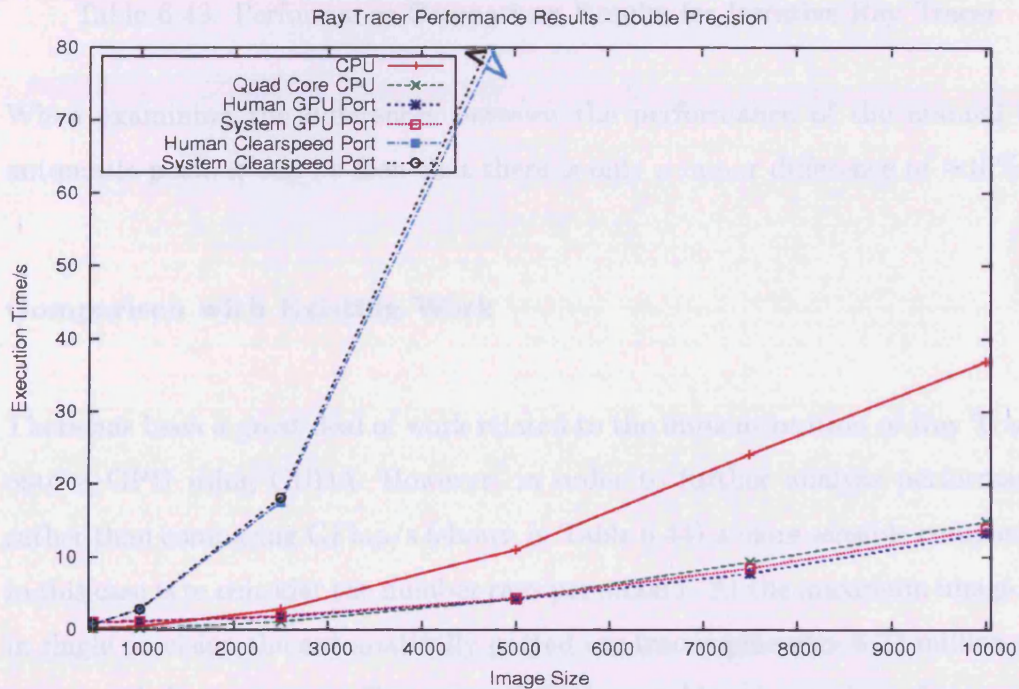


Figure 6.25: Graph of Double Precision Execution Times for Iterative Ray Tracer.

Data Size n	Single Precision Performance			Double Precision Performance		
	GPU	ClearSpeed	Optimal Device	GPU	ClearSpeed	Optimal Device
500x500	0.15x	0.23x	QCPU	0.15x	0.22x	QCPU
1000x1000	0.16x	0.05x	QCPU	0.17x	0.07x	QCPU
2500x2500	0.62x	0.06x	QCPU	0.58x	0.06x	QCPU
5000x5000	1.09x	0.05x	GPU	0.96x	0.05x	QCPU
7500x7500	1.22x	0.05x	GPU	1.10x	✗	GPU
10000x10000	1.28x	✗	GPU	1.06x	✗	GPU

Table 6.43: Performance Comparison Results for Iterative Ray Tracer

When examining the differences between the performance of the manual and automatic ports it can be seen that there is only a minor difference of $\approx 10\%$.

Comparison with Existing Work

There has been a great deal of work related to the implementation of Ray Tracers onto a GPU using CUDA. However, in order to further analyse performance, rather than comparing GFlop/s (shown in Table 6.44) a more sensible comparison in this case is to consider the number rays per second. At the maximum image size in single precision the automatically ported ray tracer generates 8.23 million rays per second. In comparison Popov et al[114] have achieved a peak performance of over 16 million rays per second, using an older GPU (NVIDIA 8800 GTX with 518GFlop/s peak).

The main performance bottleneck in the automatic port is not memory bandwidth but instead a far less severe manifestation of the problem that badly effected the ray tracer's performance on ClearSpeed. As threads are grouped together into warps, performance bottlenecks occur when only one thread within a warp is still executing because for example the ray it represents was reflected. This is known

as “warp divergence” and it causes the entire warp to continue occupying the multiprocessor when in reality only one thread is doing useful computation. In their paper Popov et al have achieved much better performance largely because they have used a much more sophisticated ray tracing algorithm, centred around utilising a stackless traversal of kd-trees[114].

Device	Peak Performance GFlop/s	Peak Memory Bandwidth MB/S	Peak Transfer Rate MB/S
Single Precision			
GPU	32.32	0.54	0.84
ClearSpeed	0.11	0	0.11
Double Precision			
GPU	26.57	0.9	0.93
ClearSpeed	0.22	0.01	0.51

Table 6.44: Ray Tracing Application Peak Performance

6.4 Classification of Known Applications

One final test that can be conducted, is to determine if the tree that has been constructed is able to predict applications that it has already seen. When attempting this, the system will attempt to classify all 216 instances that the system has already seen. When this is done the system correctly classifies 215 instances and makes a single error; a success rate of over 99%.

The only misclassification that occurred at this point is that the system wrongly predicted that the GPU will provide the best performance for the 2000x2000 dataset in the double precision GEMM seed application. Examining this prediction error it can be seen from the results presented in Table 6.9 that this application displays an odd characteristic in that the GPU outperforms the quad-

core CPU by a small amount only in this particular case. Additionally when examining Table 6.8, it can be seen that in the single precision version when using the 2000x2000 sized data-set the application performs better on the GPU but only by a very small amount (≈ 0.5 seconds). From this, it is apparent that the similar characteristics shared by the single and double precision versions of the application and the very small performance differences in question are what have caused this error. This occurred because for two very similar kernels, one executed optimally on one device and one executed optimally on another. This means that there was no way for the decision tree algorithm to divide the performance data in such a way that these differences could be expressed unambiguously.

6.5 Chapter Summary

This chapter has firstly described the performance of several seed applications. The performance results from these seed applications were used to build an initial classification model that was then used to start the following process:

- Predict which device an application will need to execute on in order to provide best performance.
- Port the application to that device.
- Gather performance data for the application on all devices.

This process was carried out for three further applications, with each application using the updated classification model that was produced by executing the previous application.

The results collected have revealed several important points:

-
- Automatic porting performs favourably when compared to a single core CPU. However, compared to a quad-core CPU the results are not as promising.
 - In three out of the seven applications (N-Body, Monte Carlo and Ray Tracing) automatically ported code outperforms the quad-core CPU. In the remaining four the quad-core CPU outperforms the automatically ported code.
 - With the exception of the Monte Carlo example, the GPU performs much better than ClearSpeed.
 - The system is able to successfully modify the classification model based on the performance data collected.
 - The more accurate the predictions that are made, the less the performance model changes. This leads to the notion that eventually, once sufficient examples have been executed, the performance model will stabilise.
 - The predictions made by the system, although not perfect, have an overall success rate of 82%.
 - That the difference between hand ported and automatically ported code for the GPU is, on average, $\approx 8\%$ and for ClearSpeed is $\approx 30\%$. However, this value considers all results, and if the N-Body simulation example is disregarded, then the value improves to $\approx 12\%$.
 - That the automatically generated code presented in this chapter, while able to outperform the quad-core CPU in certain applications, is significantly slower when compared against the optimised re-factored codes described elsewhere.
 - The majority of the automatically generated applications are severely memory bandwidth limited and it is this that is the main obstacle to achieving better

performance. This is shown in Table 6.45 which shows which factor limits each application's performance.

– Another problem encountered is that of warp divergence, which commonly occurs in applications where multiple threads executing in SIMD need to perform differing tasks.

Application	Processing Bound	Memory Bound	Comments
Sobel	×	✓	
GEMM	×	✓	
N-Body	×	✓	
Monte Carlo	×	✓	Possibility of warp divergence, but not a major factor in this case
FFT	×	✓	
Canny	×	✓	
Ray Tracer	×	×	Warp Divergence encountered

Table 6.45: Summary of factors limiting applications performance

Application	Device	Highest Speedup	
		Single Core	Quad-Core
Sobel	GPU Single Precision	1.33x	0.89x
GEMM	GPU Double Precision	3.00x	1.12x
N-Body	GPU Single Precision	65.57x	16.60x
Monte Carlo	ClearSpeed Double Precision	64.39x	16.09x
FFT	GPU Single Precision	1.81x	1.17x
Canny	GPU Single Precision	2.22x	0.88x
Ray Tracer	GPU Single Precision	3.05x	1.28x

Table 6.46: Speed-ups achieved by executing application on an acceleration device.

Examining the overall performance of all applications (shown in Table 6.46), it is obvious that, for the applications tested so far, the optimum performance is found either with the quad-core CPU or the GPU. However, in a wider environment consisting of other acceleration devices and other versions of existing devices i.e.

a NVIDIA FERMI GPU, the decision on which device to use would not be so simple. It is expected that the addition of FPGAs, CELL Accelerators, other models of GPUs and other multi core devices will provide considerable diversity, making the ability for a device to be selected automatically, possibly even without the user knowing or caring what device is being used, highly desirable.

When examining the performance increases that have been achieved, it is promising to note that speed up has been achieved in all applications compared to the single-core CPU and in three out of seven applications when compared to the quad-core CPU. Even though, the improvement is far less than the “headline” figures that have been publicised, the speed-ups that have been achieved are in essence free and require only that the input source code be passed through a different compiler (the porting system). It should also be noted that the speed-up figures given are for the entire execution of the application and not just for isolated kernels. However, there are still further optimisations that can be examined in terms of improving the performance of the ported code, and these are discussed in Chapter 8.

All of this has shown that the system has performed well in terms of device selection and, when considering performance of the ported code, it has provided performance improvement for a subset of the applications considered. For the other applications the automatically ported code is unable to outperform the quad-core CPU mainly because the algorithm that has been ported is unable to fully utilise the architecture of the GPU, largely due to the way in which these applications access memory. In order to further improve performance optimisations or re-factoring of the applications is needed.

It is expected that, despite the GPUs current dominance of the field, multi and many core devices acting as co-processors will continue to be developed and as hardware competition intensifies, the ability to automatically select a device and

then produce code for it will be essential to novice users that desire to take advantage of the performance improvements offered by these devices.

Chapter 7

Conclusions

7.1 Introduction

This chapter will firstly present the final thoughts on the work presented in this thesis and will evaluate the results that have been acquired, against the criteria set out in Chapter 1. Secondly, this chapter will compare the system that has been developed to test the hypothesis against what are deemed the two closest competing systems. This will analyse the relative strengths and weaknesses of the developed system against those of these products. This chapter will then summarise the lessons that have been learnt when working with the two acceleration devices that have been considered in this thesis and draw some conclusions as to how they compare. Finally, this chapter will briefly evaluate the successes and failures of the work overall.

7.2 Research Hypothesis

The research hypothesis that was stated in Chapter 1 is:

It is possible to construct a self-modifying and expandable automatic code porting system that can, based on heuristics, select the most appropriate application acceleration device and provide comparable performance to that achievable by an experienced human programmer.

In order to validate this hypothesis it will be broken down into sections. Each section will then be validated:

Self-modifying: Chapter 6 has shown that the system is able to modify its own internal classification model based on experience acquired from executing applications. Figures 6.10, 6.13, 6.18 and 6.23 show a series of decision trees built from the performance database. As explained in Chapter 5 this performance database is automatically augmented by the system using an algorithm that, in essence, finds and fills gaps in the performance database.

Expandable: Chapter 5 has shown how the system will expand to encompass additional devices once they are added. This is demonstrated as part of the self-modification of the system, as, in essence, the addition of new devices is a special case of the same gap filling algorithm but with a far larger amount of data that must be gathered.

Based on heuristics, select the most appropriate application acceleration device: Chapter 6 has shown that, once a set of seed applications are fed into the system, it can make predictions on both seen and unseen kernels across a variety of devices. It has been found that when the system is making predictions on kernels that it has previous seen, then the accuracy is, as expected, very high

at $\approx 99\%$. When the system is making predictions based on unseen kernels the accuracy is lower at $\approx 82\%$. This is a significantly better than randomly guessing the correct device.

Provide comparable performance to that achievable by an experienced human programmer: Chapter 6 has shown that, in the majority of cases, the system is able to produce code that is within $\approx 8\%$ for the GPU and within $\approx 12\%$ (excluding one exceptionally poor case) for ClearSpeed, of that produced by an experienced human programmer.

This test has, however, assumed that the hand ported code is also a direct port of the input code and the application is not “re-factored” to achieve higher performance. This is to enable us to test the code generation ability of the system against a human programmer without being subject to a human’s ability to intelligently reconstruct an application that performs the same task but in a different form.

There are improvements that can be made regarding these tests. Firstly, a different programmer than the developer should be used to develop the manual ports of applications. This would prevent the possibility of the introduction of any bias, although, it should be stressed, that the manual ports that have been produced are to the best of the developers ability, ensuring the best possible performance is achieved within the restrictions that have been given.

Secondly, the introduction of additional programmers will provide a fairer overall comparison. Ideally, the developers should not have knowledge of how the system works but they would need detailed knowledge and skills in programming the respective devices.

The main problem with carrying out this experiment is locating developers with

the required skills. This is important because if the developers chosen are novices, then the performance of the manual ports would almost certainly be worse than has been presented in this thesis. One possible time saving measure that could be introduced, is to use the developed system to provide an initial port of an application to the developers from which they can improve the code. Even with this possible time saving measure, it was still deemed impractical within the scope of this doctoral program, due to the lack of other developers, to carry out the experiment at this time.

When comparing the automatically generated code against re-factored versions of the same algorithm, Chapter 6 shows that automatically generated code does not compete favourable against tuned implementations, such as those that have been discussed in Section 2.6 of Chapter 2. This is not a surprising outcome but it is encouraging to see that automatically generated code is able to consistently outperform a single-core CPU and, in certain cases, a quad-core CPU.

7.3 Contributions

Contribution 1: A novel distributed system and architecture that is able to analyse and port input applications to an acceleration device and, with a reasonable success rate, predict the most appropriate device for the application concerned.

This contribution is drawn directly from the hypothesis and it has been shown above that the system that has been constructed is able to both port code with good efficiency and make predictions of reasonable accuracy. No known system of this type has been previously published and this is supported by the publication of two peer reviewed papers and the comments received from the referees.

Contribution 2: Demonstration that the system is able to modify itself, in that it is able, from experience, to adapt the model that is used to select an acceleration device and that it is able to adapt to the introduction of the new device's, or improved versions of existing devices.

This contribution is also drawn directly from the hypothesis and we have shown in the previous section that the system is not only able to modify itself based on data from running existing, and new applications, but it is also modify itself based on the introduction of new devices, as this process is a special case of the system self-modification mechanism.

Contribution 3: The ability to demonstrate, through the use of well understood and developed machine learning techniques, a set of explicit parameters and features that can be used to describe the selection of an appropriate acceleration device.

This contribution justifies the inherent value and the expressive power of the data that the system collects. The decision tree method of classification was chosen mainly because of its ability to be easily understood by humans. This results in the decision trees produced by the system being very valuable in their own right. Even if a user did not possess any devices, they could examine such a decision tree along with their code and make a decision, without the need for experimentation, as to which device would be appropriate to select. This is exhibited in Figure 6.23, which express the criteria for choosing an optimal device from all the devices our system has used (Quad-Core CPU, GPU and ClearSpeed).

Contribution 4: The provision of a route to application acceleration to novice users. This may be the porting of an application, generating an efficient initial port from which further performance improvements can be made by experienced human programmers, or determining in a quick and simple manner, whether the

users application is suitable for acceleration.

The increasing reliance on parallel technologies and the introduction of hybrid systems has made HPC far more complex for novice users. This thesis has shown that the generation of automatically ported code is a viable and efficient strategy in certain circumstances. However, in other circumstances the performance of automatically generated code is often inferior to existing libraries and re-factored version of the algorithm.

The analysis of the performance of the applications in Chapter 6 has shown that they are all (apart from the Ray Tracer) memory bound and, in order for their performance to improve, automatically generated code needs to map better to the device's memory architecture. Something which the GPU and ClearSpeed code generators developed in the thesis are not yet able to do.

Table 7.1 shows the percentage of peak performance that each application has achieved, where the peak performance is the highest performance that has been found on a comparable device. It should be noted that in certain cases that no comparable figures could be found, meaning that a complete comparison against all applications in single and double precision is not possible. Nevertheless, the table does allow trends to be identified. Examining Table 7.1 it can be seen that the relative performance for the Sobel, GEMM and FFT applications is low, but the Monte-Carlo and N-Body simulation provide a far better comparison, achieving 31% and 39% of peak performance respectively.

Even though automatically generated code is not, in many cases, able to compete with the performance provided by a re-factored and tuned application that has been ported to the device, the system that has been developed could still have significant impact in providing an easier route to the use of application acceleration devices within future HPC systems, especially for novice users. The

Application	Re-factored Algorithm	% of Peak Performance Achieved	
		Single Precision	Double Precision
Sobel	Brandvik et al[21]	2.6%	Not Available
GEMM	Nath et al[96]	1.1%	5.2%
N-Body	Nyland et al[109]	39.2%	Not Available
Monte-Carlo	ClearSpeed[37]	27%	31%
FFT	Govindaraju et al[59]	0.7%	Not Available

Table 7.1: Percentage of Peak Performance Achieved by Automatically Generated Code

developed systems strength lies in its ability to provide an essentially intellectually effort free route of access to the use of an acceleration device.

This ability can be leveraged by end-users in one of several ways. The system could be used to port applications, determine if an application is viable for acceleration or used as a rapid prototyping system. Take for example the situation where a user wishes to develop the most efficient possible port of their application. They could utilise the porting system to generate an initial port of the code and that code could, from the output produced by the system, be incrementally made more efficient by expert human intervention. Another possible example would be the utilisation of the system by a novice user looking to purchase an acceleration device. The user would use the system to predict on which device their application gives the best performance, this enables them to make a more informed decision prior their purchase.

7.4 Relation to Current Work

This section will outline the strengths and weaknesses of the developed system when compared to what are deemed the closest existing approaches: HMPP and the PGI Accelerator Compiler. Both of these tools were discussed in detail in

Section 2.7 in Chapter 2.

7.4.1 HMPP

The HMPP system provides the ability for the programmer to augment their code with OpenMP like directives. Once augmented, the HMPP system allows the code identified by these directives to be executed on acceleration devices. The following outlines the various strengths and weaknesses compared to the developed automatic self-modifying application porting system.

Strengths:

- Supports Fortran in addition to C,
- Supports OpenCL as an additional back-end,
- OpenMP like directives will be familiar to many developers, and allow the expression of details that are not originally expressible in the input language.

Weaknesses:

- User must select which kernels to execute by way of directives added to the code,
- Selection of device is based on the device availability on the host node.

7.4.2 PGI Accelerator Compiler

The Accelerator Compiler from PGI also allows the programmer, using a series of OpenMP like directives, to indicate which statements are to be accelerated.

The following outlines the various strengths and weaknesses compared to the developed automatic self-modifying application porting system.

Strengths:

- Supports Fortran in addition to C,
- OpenMP like directives will be familiar to many developers, and allow the expression of details that are not originally expressible in the input language,
- PGI Compiler is able to override users choices if array dependencies are detected,
- PGI Compiler is able to detect performance bottlenecks and advise the user appropriately,
- PGI Compiler is able to undertake a rich portfolio of optimisations such as re-ordering of nested loops.

Weaknesses:

- User must select which kernels to execute by way of directives added to the code,
- Only one back-end supported, so device selected is irrelevant.

7.4.3 Conclusions

The above analysis of what is deemed the two most directly competitive systems leads to the following conclusions:

- The PGI Compiler has superior front end parsing abilities,
- Neither of the two systems provide mechanisms for selecting the device and they generally make the assumption that the compiled binary will be executed on single device systems.
- The selection of kernels for execution is done by the programmer by way of inserting directives. The developed automatic self-modifying application porting system eliminates this need by automatically selecting appropriate kernels from the code and then determining if they are suitable for execution.

7.5 Evaluation of Metrics

One of the key features of the system is its ability to select an appropriate device for an application based on a series of metrics. The metrics that have been utilised so far are:

- The highest precision data-type that is used by the application.
- A count of mathematical operations (intensity).
- A count of the number of memory accesses (read and write).
- A count of branching that occurs.
- The number of iterations of the kernel that are performed.
- Size of data that must be loaded to/from the device.

These metrics have been selected based what on has been determined, from experience, to be the key program features relating to the performance of applications on acceleration devices. The system itself, by its use of a decision tree classification model, provides its own internal validation of these metrics and any metrics that are not significant will be factored out by the decision tree induction algorithm.

The surprising result of this is that the only metric that is unused is the highest data precision metric (i.e. single or double precision) and this metric does not have any effect on the selection of the device within the decision model that has so far been constructed. This is shown by the results and the decision trees presented in Chapter 6. However, it should be noted that when additional devices are added to the system, this is expected to change. An example of this would be the introduction of a FPGA acceleration device, which presently strongly favours integer and single precision arithmetic over double precision.

A key point that was realised in the development of the system is that as much as possible the metrics should be independent and orthogonal to each other. This is illustrated in the case of the data loaded to and from the device. If these two figures are treated individually then they make implications about the other metrics. i.e. if data out is much lower than data in, then it implies that the amount of memory written and the amount of memory read by each iteration will have a similar relationship.

From the work conducted so far, it is believed that appropriate metrics have been selected and used, and in many ways it is better to have too many metrics within the system and allow the classification model to eliminate them, than to have too few.

7.6 ClearSpeed vs NVIDIA GPU

One of the main considerations when comparing ClearSpeed and NVIDIA GPUs is the comparative ages of the technologies. While the ClearSpeed device's most recent iteration was released in 2008 [87], the fundamental architecture has not changed since 2005. The GPU architecture that has been used was first designed in 2006 [45]. However, the CUDA system itself was not released until 2007 [46]. Finally the specific model of GPU that has been used in this thesis was released in 2008 [101].

This illustrated that in reality the ClearSpeed system is, in terms of design, approximately two years behind NVIDIA. However, development on the ClearSpeed accelerator, as far as can be seen, has all but stopped.

This lack of progress is apparent because, in all except one case, ClearSpeed is completely unable to compete with the GPU in terms of performance. The lack of performance presented by ClearSpeed is due to several reasons:

- CUDA has developed a novel method of hiding memory latency by the rapid context switching of groups of threads known as warps, whereas ClearSpeed leaves this to the developer.
- CUDA only executes small numbers of threads in SIMD, where ClearSpeed executes $\frac{NoThreads}{2}$ in SIMD. This enables CUDA to overcome the problem of uneven load balancing between threads as shown in the Ray Tracing example.
- CUDA has a larger on-board memory size.
- Each GPU Streaming Core has a clock frequency of 1.3 GHz while each ClearSpeed Processing Element has a clock frequency of 250MHz. However it

should be noted that a ClearSpeed board consists of 192 Processing elements whereas the GPU consists of 240 streaming cores.

- Each Streaming Multiprocessor has a shared memory of 16Kbytes whereas each ClearSpeed core only has a local memory of 6Kbytes.
- The programming level of C_N is at a noticeably lower level than CUDA and when programming in C_N you must manually handle data movement between the device's main memory and the processing elements. The CUDA system does this automatically and also features methods for reducing the memory transfers to the absolute minimum. This difference in API level can be illustrated by noting that the ClearSpeed back-end totals 4000 lines of code whereas the CUDA back end totals just over 1200.

7.7 Evaluation

Looking back at the work that has been conducted, there are several things that could be improved if the project was to be repeated or extra time was available:

- The one major problem throughout the project has been the poor comparative performance of the ClearSpeed accelerator. When the project started, GPGPUs were in the early days of development and when the initial work was carried out with the ClearSpeed accelerator it was competitive with GPU devices that were available at that time. However, since then the GPU has gone through several iterations, whereas ClearSpeed has not released any new models. This had led to ClearSpeed simply being unable to compete with the GPU. When this was realised, it would have been desirable to use another device. However,

there simply were no other devices that could be used within the time-scale of the project.

– One possible weakness in the proof of the hypothesis that has been presented is the fact that only a single programmer has been used, whereas, in an ideal world, a large survey set of experienced programmers would have been used. Unfortunately, due to the unfamiliarity of many developers with CUDA and, especially, ClearSpeed programming this simply was not possible. Due to the time that would be required to port these applications, and the level of experienced required, the only feasible way to conduct this particular comparison more rigorously would be to employ a group of expert programmers to port the applications.

– Examining again the seven dwarfs model outlined by Asanovic et al [14] this thesis has outlined examples from five out of the seven dwarfs. It has not tackled applications from the Unstructured Grids or Sparse Linear Algebra categories. However, it is anticipated that the system will functional equally well for application from these two dwarfs and including them would only be necessary for completeness.

– Initial results when comparing against a single core CPU were very promising. However, once quad-core CPU performance was included the results were less promising. When comparing against the single-core CPU all applications achieved performance improvement once ported to the GPU but when comparing against

the quad-core CPU, three out of the seven applications (N-Body, Monte-Carlo and Ray Tracing) achieved performance improvement on the GPU, with the other four applications performing optimally on the quad-core CPU.

7.8 Chapter Summary

This chapter has analysed the overall success of the system. It has shown that although the system is not able to predict unseen applications with complete accuracy it is able to make predictions with a success rate of over 80%.

This chapter has also shown that the system is able to produce code with a performance that compares favourably to that achieved by an experienced human programmer, assuming no re-factoring takes place. However, when comparing automatically generated code against that of re-factored code, large performance gaps have been shown. These show the importance of being able to adapt the algorithm in use to suit the architecture of the device, this means that end-users should endeavour to leverage on libraries providing optimised versions of their code whenever possible. In situations where optimized versions of an algorithm are not available, automatic porting can achieve for certain applications virtually effort free performance improvement and also enable end-users to utilise the generated code as a platform to aid and inform them in the construction of their own optimised version of the algorithm.

This chapter has also described how automatically generated code for the GPU outperformed the single-core CPU in all applications and the quad-core CPU in certain cases. However, even though based on final decision tree constructed by the application porting system system, the GPU would only be selected to accelerate a sub-set of the kernels encountered, the system is

expandable. This would allow the addition of new devices such as a NVIDIA Tesla C2070(FERMI)[106], which will then result in changes to the decision model.

Despite this, this chapter has shown that the hypothesis proposed in Chapter 1 is true and that it is possible to construct a system that meets these criteria. However, in order for such a system to be commercially viable it is anticipated that further work must be done. Chapter 8 discusses this and suggests one possible method of deployment for the system.

The system that has been described in this thesis can also be compared to the ideal system presented in Chapter 2, Figure 2.10. Examining the key factors of that system it can be seen that each of the requirements outlined has been tackled, although for this ideal system to be constructed, each aspect would have to be taken to the limit:

The ability to port code with no user intervention: This requirement has been met, however, problems can arise when the lower level API of the device requires information that is not expressible in the input language used. There are however extensions to this functionality that can add to the performance of automatically generated code and these will be discussed in Chapter 8.

The ability to select the device to execute the application: This requirement has been met, although the accuracy of the decisions that are made are dependent on the amount and accuracy of the performance data that has been collected.

The ability to target all devices: Back-ends for two devices have so far been implemented. In order to fulfil this requirement an ideal system would need to have back-ends for every acceleration device type.

The ability to operate on all known applications: This requirement relates

strongly to the power of the client. The more languages that are supported and the better at code analysis that the front end client is, the more applications that the system can process. Currently a front-end for the ANSI C language has been implemented. However, it is noted that this could be improved by adding additional front end analysis, such as further loop dependency analysis and other techniques, such as loop-unrolling and the analysis of nested loops used by the PGI accelerator compiler (described in Chapter 2).

Implementing an ideal system, as described above, is in all likelihood impractical, but it is a useful comparison to the porting system that has been developed. It is my belief that even though there is a great deal of work to be done in order to advance the development of the porting system, all the intellectually difficult areas have been tackled within this thesis.

As a final thought, it is anticipated that this system will be of benefit to those developers who are unfamiliar with acceleration devices and simply wish to acquire the best possible performance for their application. It is also anticipated that users of legacy codes will be especially interested. However, going forward, it is also obvious that, in order for the boundaries of computing to be pushed forward, parallel thinking, as opposed to sequential thinking must become the norm, if new hybrid HPC system consisting of collections of many-core and multi-core systems are to be fully leveraged.

Chapter 8

Further Work

8.1 Introduction

This chapter will consider the work that has been described in this thesis with a view for continuation of the research.

This chapter will only consider improvements from a research perspective and will not consider purely engineering challenges such as:

- The addition of new back-end devices.
- The addition of new front ends, such as Fortran90.
- The implementation of additional loop dependency checking.

The main ideas that will be discussed in this chapter include:

- The ability to make decisions based on characteristics other than pure performance,
- Adding the ability to support larger data-sets,

- Supporting Multi-Card Accelerators,
- Intelligent Scheduling,
- Code Optimisations,
- Mapping of code to pre-existing libraries.
- Possible applications of the porting system regarding cloud computing,
- Ideas related to a process for an adaptive porting system.

8.2 Making decisions based on other factors

Currently the porting system makes its decisions based purely on performance data. However, there are other factors to consider:

Power: If several acceleration devices are able to provide acceleration for an application, then a user may wish to select the most power efficient device for their application. This method could use a measurement such as *FLOPS/WATT*, or a measure of the carbon footprint that is produced in order to power the machines. Both of these metrics could be used in addition to performance to decide the optimum device.

Financial: It is feasible that the porting system, along with appropriate performance data, could be used to select an appropriate device for an application prior to the device being purchased. In this case then the following measure of financial outlay and performance achievement could be used:

$$\frac{ExecutionTime_{CPU} - ExecutionTime_{Device}}{Cost_{CheapestDevice} - Cost_{Device}}$$

Where cost is the total cost of ownership for the life-cycle of the device.

This would compare the ratio of the differences in performance to that of cost and would allow some judgement to be made as to how cost efficient each device is, i.e. it would not normally be acceptable to spend many thousands of pounds extra to achieve a minor performance gain.

8.3 Code Optimisations

There are several program optimisations for both CUDA and ClearSpeed which can be explored and, if possible, added to the system.

8.3.1 ClearSpeed:

The main optimisation that can be added to ClearSpeed is to determine if there is a feasible method for exploiting ClearSpeed's Swizzle operation. The Swizzle operation allows register to register data transfers between neighbouring processing elements. However, the lower level nature of this operation means that the system would need to be able to specifically detect features of the input code that would map onto the Swizzle operation. This is a difficult problem to solve, as in many cases the information required to take advantage of Swizzle is not expressible in the C language.

8.3.2 CUDA

There are two viable optimisations that could be investigated for addition to the CUDA back-end that has been constructed:

Optimisation of Global Memory Access

In CUDA the memory accesses of each half-warp are coalesced by the device into as few transfers as possible when certain requirements are met [46]. The requirements for the device to coalesce memory access are complex, but it may be possible for the system to perform a series of program transformations to facilitate the device's ability to coalesce global memory access. Baskaran et al [15] has already done work in this field, but they have, so far, limited their work to focus on optimisations of affine loop nests. Additionally, Ueng et al [131] have produced CUDA-Lite, which is software that allows the optimisation of global memory accesses using a series of programmer annotations. It would be of interest to see if either of these techniques can be adapted or expanded to provide this valuable optimisation with the system that has been described in this thesis.

Making increased use of shared memory

It is possible to reduce the number of loads from global memory to the memory on the individual processing units by utilising each multiprocessor's shared memory. In order to do this the system would need to detect common memory accesses between threads and add additional code before the execution of the thread begins to copy of the shared accesses to the multiprocessors shared memory. This could be expanded to also allow the CUDA back-end to utilise texture memory. This

would be beneficial in cases where there is 2D locality in fetches from memory [102]. However, the major disadvantage is that texture memory is read only and the performance of memory fetches is often uncertain if certain requirements (i.e. 2D spatial locality) are not met. This would mean that the system would need to detect when using texture memory is appropriate, and only then generate the appropriate code.

8.4 Supporting larger data-sets

One of the problems that has been encountered while carrying out this work was that, especially on devices with limited memory such as ClearSpeed, that the device memory was simply insufficient to contain the input/output data-sets that the application required.

This problem could be solved by doing the following:

- Detecting the memory requirements of each iteration of the kernel.
- Determine the maximum number of iterations that each device can fit in memory X .
- This means that the entire execution process on the device will need to repeat $\lceil \frac{TotalIterations}{X} \rceil$ times.
- Balance this figure so the load is evenly distributed.

This, however, will not always be possible, as in some cases it is simply not possible to determine the memory requirements of a kernel in advance of its execution. In these cases there are two options that could be investigated:

- Accept that if the entire application cannot fit into memory then the application cannot be executed on the device,
- Provide a method for the programmer to guide the system in determining what the memory requirements of each iteration of the kernel are, such as annotations or interaction during the compilation process.

8.5 Supporting Multi-Cards Accelerators

The system that has been developed to date treats each acceleration device as a separate entity, even if several devices are connected to one host node. Possible further work, would be the added ability to treat the set of all devices connected to one host node as one single device when necessary. Additionally, this could be expanded to also treat other facilities available on the host node, i.e. a multi-core CPU, as additional devices to assist with accelerating the application.

This would lead to the addition of several new “composite” devices within the system. As these larger devices would only be applicable in cases where the input application can be separated to such an extent to allow different segments of it to execute on each card. The main obstacle that would need to be overcome is that each device within the “composite” device would not have access to the device memory of other devices. Secondly, the system would have to view the composite device both as a device consisting of X cards and X individual devices consisting of a single card. The reason for this is that not all applications will benefit from being executed on a “composite” device and having such a large amount of hardware sitting idle would be a tremendous waste of resources. This would also raise the issue of scheduling and the problem of ensuring that all cards within a “composite” device are kept free to enable an application to execute.

8.6 Scheduling

Another possible improvement that could be made to the system is the addition of an intelligent scheduler. Such a scheduler would attempt to solve the following issue that would arise in a production system:

If a user has to wait X minutes for device A . Then as long as device B is available and provides performance within $X - 1$ minutes of device A , then device B would be the most efficient device to use.

The improvement would be necessary to ensure that a system consisting of many application acceleration devices was truly adaptive to the application load that would be placed upon it.

8.7 Mapping code to computation libraries

A key point that has been identified in the course of this work is that automatically generated code, generally speaking, cannot compete with the optimised algorithms often present in computational libraries. This issue has led to the development of the idea of mapping input source code to that of an existing library and there are two circumstances in which this idea could be applied.

Firstly, to enable the mapping of CPU based libraries to those present on an acceleration device i.e. mapping FFTW(CPU library) to CuFFT(GPU library). In developing this approach care would need to be taken in ensuring that the method of calling the CPU library can be translated onto that required by the device's library and that either the data format used by the libraries are compatible or code can be generated to enable a conversion to take place.

The second, and far more complex scenario, requires the application porting system to recognise known algorithms within a kernel. This would involve the application porting system extracting the algorithmic form of a kernel and then attempting to match it to a database of algorithmic forms, each with an associated library function call. However, before such an approach could be developed many key problems would need to be solved, such as:

- Mapping of input and output data from the format used in the ported code to that of the library.
- Implementing sufficiently permissive matching of user's code to algorithmic forms to allow successful matching despite differing programming styles.
- Implementing sufficiently restrictive algorithm detection to ensure that the correct library call is chosen.
- Many computation libraries in order to achieve the best performance require extra information regarding the characteristics of the input data-set in advance. This information may not be available through automatic analysis.

8.8 Cloud computing

One of the most interesting areas of future expansion for the system is making it available within a cloud computing environment in the manner of "Software as a Service". In order for the system to function in this environment, the main addition that would be needed is the implementation of a scheduler as discussed previously.

An example of a possible architecture for the system within a cloud is shown in Figure 8.1.

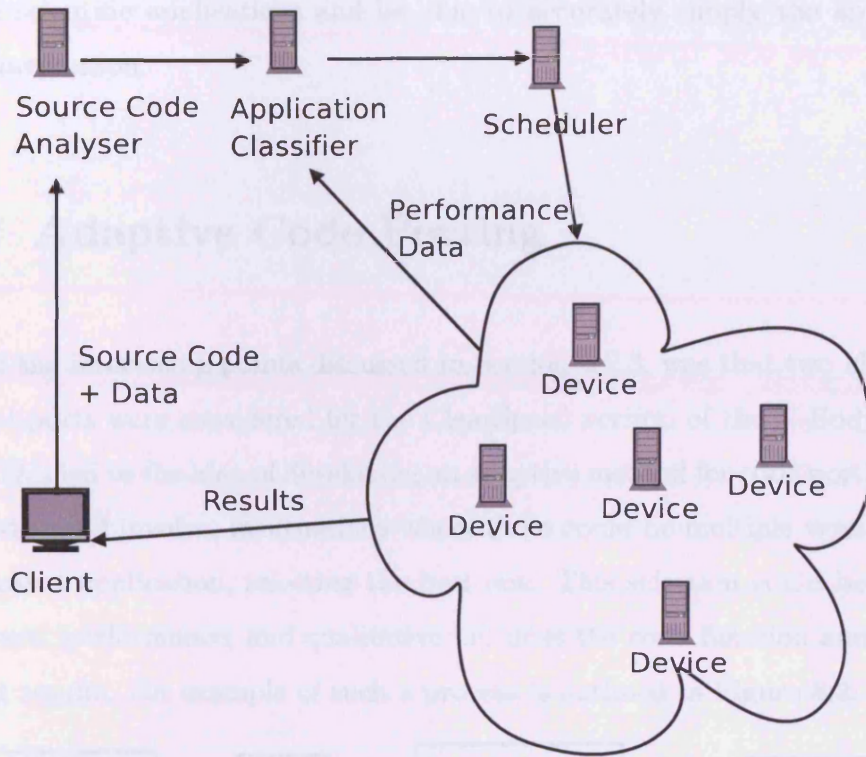


Figure 8.1: Cloud Computing.

Such a system could provide a subset of services to end users in addition to the overall functionality that has been outlined in this thesis. These services could include:

- A system that recommends a device to execute a given application.
- A system that ports code i.e. a rapid prototyping system.

One of the main advantages of a cloud computing system is that such a system, given sufficient usage, would be able to rapidly acquire a large amount of

performance data. This would mean that when a user supplies an application, the system will have already seen many of the common kernels that occur within scientific applications and be able to accurately supply the appropriate recommendation.

8.9 Adaptive Code Porting

One of the interesting points discussed in Section 6.2.3, was that two alternative manual ports were considered for the ClearSpeed version of the N-Body simulation. This led to the idea of developing an adaptive method for code porting. This method would involve, in situations where there could be multiple versions of an accelerated application, selecting the best one. This selection could be made in two ways: performance, and qualitative i.e. does the code function and produce correct results. An example of such a process is outlined in Figure 8.2.

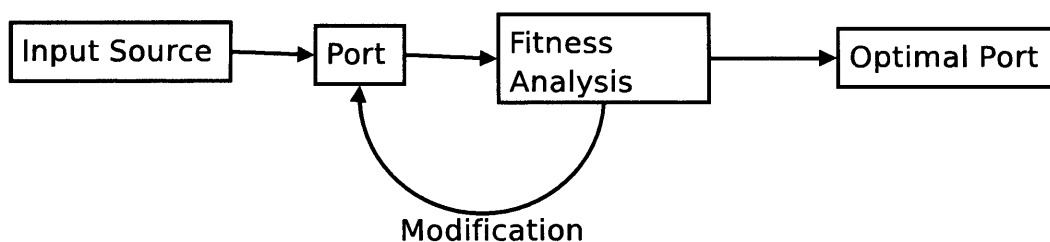


Figure 8.2: Adaptive Porting.

Examples of when this process could be useful would include:

- Deciding if a ClearSpeed port should divide the input data-set between memory chips or if it should duplicate data between them.
- Deciding if a kernel should use double buffering or not.

– If there were a set of optimisations that could be applied, i.e. use of the Swazzle operator, these optimisation could be applied and validated. If the code functions and gives correct results, the optimisations can be used, if not then they can be discarded.

8.10 Chapter Summary

This chapter has outlined several improvements that could be made to the automatic self-modifying application porting system. The culmination of these improvements is the development of the system as “Software as a Service” within a cloud computing system. This future work would enable the provision of application acceleration to the wider computing community without the requirement for detailed knowledge of the underlying device that executes the application.

One of the implications of this, is that the availability of different device types will need to be carefully managed. This would mean that, given a system with a finite number of devices, not every user would be able to execute their application on the optimum device without waiting. This could be tackled by the introduction of pricing mechanism and a rare device that provides excellent performance would undoubtedly cost more to utilise than other devices.

In addition to these improvements relative to cloud computing, other work can be done such as the ability to handle applications with data requirements larger than the device’s memory. This process is reasonably easy in cases where the system can identify memory requirements at compile time, but, when this is not possible, further work is needed to investigate an appropriate solution.

Additional code optimisations can also be investigated and added to the system. In terms of the ClearSpeed accelerator, the use of Swizzle needs to be investigated as does how the input could be used to “guide” the system in using this lower level operation. In terms of CUDA, the most important optimisation is the implementation of further code transforms to ensure that global memory access is, where possible, always coalesced.

Finally, ideas are presented related to the idea of adaptively porting code. This means that a porting system may produce several different ports of an application, which are then analysed based on performance and quality. This enables the system to try multiple methods of producing code and perhaps different combinations of optimisations that may or may not function as expected. While only very early ideas for this are presented, it is anticipated this could be highly useful, especially on devices that possess a lower level programming model (i.e. ClearSpeed).

Bibliography

- [1] ClearSpeed Company Website. [Online] <http://www.Clearspeed.com>, [Accessed Dec 2009].
- [2] FLAGON Website. [Online] <http://sourceforge.net/apps/trac/flagon/wiki>, [Accessed February 2011].
- [3] GNU-C Library Manual. [Online] <http://www.gnu.org/software/libc/manual>, [Accessed June 2010].
- [4] IBM Website. [Online] <http://www.ibm.com>, [Accessed February 2011].
- [5] LibSH Project Website. [Online] <http://www.libsh.org>, [Accessed May 2010].
- [6] Magma Library Website. [Online] <http://icl.cs.utk.edu/magma/>, [Accessed February 2011].
- [7] Nallatech Company Website. [Online] <http://www.nallatech.com>, [Accessed Dec 09].
- [8] Turbostream. [Online] <http://www.turbostream-cfd.com/>, [Accessed March 2011].
- [9] Weka Manual. [Online] <http://www.cs.waikato.ac.nz/ml/weka/>, [Accessed June 2010].
- [10] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : Principles, Techniques, and Tools*. Reading, Mass. ; Wokingham : Addison-Wesley, 1986.

- [11] Erik Alerstam, Tomas Svensson, and Stefan Andersson-Engels. Parallel computing with graphics processing units for high-speed Monte Carlo simulation of photon migration. *Journal of Biomedical Optics*, 13, 2008.
- [12] AMD. AMD Core Math Library Website. [Online] <http://developer.amd.com/cpu/Libraries/acml/>, [Accessed February 2011].
- [13] David Andrews, Ron Sass, Erik Anderson, Jason Agron, Wesley Peck, Jim Stevens, Fabrice Baijot, and Ed Komp. Achieving Programming Model Abstractions for Reconfigurable Computing. *IEEE Transactions on Very Large Scale Integration Systems*, 16:34–44, 2008.
- [14] Krste Asanovic, Ras Bodik, Bryan Catanzaro, Joseph Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Plishker, John Shalf, Samuel Williams, and Katherine a Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical report, University of California at Berkeley, 2006.
- [15] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 225–234, 08.
- [16] Thomas Beach. Poster: An Intelligent Semi-Automatic Application Porting System for Reconfigurable Devices. *Many-core and Reconfigurable Supercomputing Conference (MRSC)*, 2008.
- [17] Thomas H Beach, Ian J Grimstead, David W Walker, and Nick J Avis. Abstraction of Programming Models Across Multi-Core and GPGPU Architectures. In *Proceedings of International Conference on Parallel Computing(PARCO)*, 2009.
- [18] Thomas Henry Beach and Nicholas J Avis. An Intelligent Semi-Automatic Application Porting System for Application Accelerators. *Conference on Computing Frontiers: Proceedings of the Combined Workshops on*

- UnConventional High Performance Computing Workshop plus Memory Access Workshop*, pages 7–10, 2009.
- [19] Pieter Bellens, Joesp M Perez, Rosa M Badio, and Jesus Labarta. CellSs: a programming model for the cell BE architecture. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 86, 2006.
- [20] David Blythe. The Direct3D 10 System. *ACM Transactions on Graphics*, 25:724–734, 2006.
- [21] Tobias Brandvik and Graham Pullan. An Accelerated 3D Navier-Stokes Solver For Flows In Turbomachines. *ASME Transactions, Journal of Turbomachinery*, 133, 2009.
- [22] Tobias Brandvik and Graham Pullan. SBLOCK: A Framework for Efficient Stencil-Based PDE Solvers on Multi-core Platforms. *Proceedings of International Conference on Computer and Information Technology (CIT) 2010*, pages 1181 – 1188, 2010.
- [23] Jens Breitbart. CuPP - A Framework for Easy CUDA Integration. In *Proceedings of the IEEE International Symposium on Parallel&Distributed Processing*, pages 1–8, 2009.
- [24] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Transactions on Graphics (TOG)*, 23:777–786, 2004.
- [25] Duncan Buell, Tarek El-Ghazawi, Kris Gaj, and Volodymyr Kindratenko. High Performance Reconfigurable Computing. *IEEE Computer*, 40:23–27, 2007.
- [26] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated Volume Rendering and Tomographic Reconstruction using Texture Mapping Hardware. In *Proceedings of the 1994 Symposium on Volume Visualization*, pages 91–131, 1994.

- [27] Timothy John Callahan. *Automatic Compilation of C for Hybrid Reconfigurable Architectures*. PhD thesis, University Of California, Berkeley, 2002.
- [28] J Canny. A Computational Approach to Edge Detection. *Readings in Computer Vision*, pages 184–203, 1986.
- [29] Allan Cante. A Review of the HPRC Industry Current Progress & Future Predictions. In *Manchester Reconfigurable Supercomputing Conference*, 2007.
- [30] Oregon Medical Laser Center. Monte Carlo Simulations. [Online] <http://omlc.ogi.edu/software/mc/>, [Accessed June 2010].
- [31] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, and Kevin Skadron. A Performance Study of General-Purpose Applications on Graphics Processors using CUDA. *Journal of Parallel and Distributed Computing*, 68:1370–1380, 2008.
- [32] Francisco Chinchilla, Todd Gamblin, Morten Sommervoll, and Jan F Prins. Parallel N-Body Simulation using GPUs. [Online] <http://wwwx.cs.unc.edu/~tgamblin/gpgp/GPGPfinalRcport.pdf>, 2004 [Accessed December 2009].
- [33] ClearSpeed. ClearSpeed Application Note: Accelerating computer assisted molecular modeling for drug design. Technical report, ClearSpeed, 2007.
- [34] ClearSpeed. ClearSpeed Application Note: Ground-Breaking Acceleration Quantum Chemical Calculations Using MOLPRO. Technical report, 2007.
- [35] ClearSpeed. ClearSpeed Introductory Programming Manual. Technical report, ClearSpeed, 2007.
- [36] ClearSpeed. CSX Processor Architecture. Technical report, ClearSpeed, 2007.
- [37] ClearSpeed. Credit Risk Analysis. Technical report, ClearSpeed, 2008.
- [38] Clearspeed. FFT Performance. [Online] <http://www.clearspeed.com/applications/>, [Accessed] January 2011.

- [39] James W. Cooley and John W Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19:297–301, 1965.
- [40] Jay L. T. Cornwall, Lee Howes, Paul H. J. Kelly, Phil Parsonage, and Bruno Nicoletti. High-Performance SIMT Code Generation in an Active Visual Effects Library. *Proceedings of the 6th ACM conference on Computing frontiers*, pages 175–184, 2009.
- [41] Convey Computer Corporation. The Convey HC-1 Computer: Architecture Overview. Technical report, Convey Computer Corporation, 2009.
- [42] IBM Corporation. Software Development Kit for Multicore Acceleration: Programmer’s Guide. Technical report, IBM Corporation, 2007.
- [43] NVIDIA Corporation. NVIDIA Tesla C1060 Computing Processor. [Online] http://www.nvidia.com/object/product_tesla_c1060_us.html, [Accessed Aug 2010].
- [44] NVIDIA Corporation. The NVIDIA Tesla S1070 Computing System. [Online] http://www.nvidia.co.uk/object/tesla_s1070_uk.html, [Accessed Aug 2010].
- [45] NVIDIA Corporation. Technical Brief: NVIDIA GeForce 8800 GPU Architecture Overview. Technical report, NVIDIA Corporation, 2006.
- [46] NVIDIA Corporation. NVIDIA CUDA Programming Guide. Technical report, NVIDIA Corporation, 2007.
- [47] NVIDIA Corporation. NVIDIA GeForce GTX 200 GPU Architectural Overview. Technical report, NVIDIA Corporation, 2008.
- [48] NVIDIA Corporation. NVIDIA’s Next Generation CUDA Compute Architecture: Fermi. Technical report, NVIDIA Corporation, 2009.
- [49] Romain Dolbeau, Stephane Bihan, and Francois Bodin. HMPP: A Hybrid Multi-core Parallel Programming Enviroment. Technical report, CAPS Enterprise, 2009.

- [50] Douglas Eadline. The Cost to Play: CUDA Programming. [Online] <http://www.linux-mag.com/id/7707>, [Accessed Aug 2010].
- [51] A. E. Eichenberger, J. K. OBrien, K. M. OBrien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using Advanced Compiler Technology to Exploit the Performance of the Cell Broadband Engine Architecture. *IBM Systems Journal*, 45:59–84, 2006.
- [52] Tal Elgar. Intel Many Integrated Core (MIC) architecture. [Online] <http://www.many-core.group.cam.ac.uk/ukgpucc2/talks/Elgar.pdf>, [Accessed January 2011].
- [53] Erich Elsen, V Vishal, Mike Houston, Vijay Pande, Pat Hanrahan, and Eric Darve. N-Body Simulations on GPUs. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.
- [54] Rahul Garg and José Nelson Amaral. Compiling Python to a Hybrid Execution Environment. *GPGPU '10: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 19–30, 2010.
- [55] Gildas Genest, Richard Chamberlain, and Robin Bruce. Programming an FPGA-based Super Computer Using a C-to-VHDL Compiler: DIME-C. In *Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems*, pages 280–286, 2007.
- [56] Anwar Ghuloum, Eric Sprangle, Jesse Fang, Gansha Wu, and Xin Zhou. Ct: A Flexible Parallel Programming Model for Tera-scale Architectures. Technical report, Intel, 2009.
- [57] Dominik Goddeke, Hilmar Wobker, Robert Strzodka, Jamaludin Mohd-Yusof, Patrick McCormick, and Stefan Turek. Co-Processor Acceleration of an Unmodified Parallel Solid Mechanics Code with FeastGPU. *International Journal of Computational Science and Engineering*, 4:254–269, 2009.

-
- [58] Mark Goresky and Andrew Klapper. Efficient Multiply-with-Carry Random Number Generators with Maximal Period. *ACM Transactions on Modeling and Computer Simulation*, 13:310–321, 2003.
- [59] Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High Performance Discrete Fourier Transforms on Graphics Processors. *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 2:1–2:12, 2008.
- [60] Khronos Group. OpenCL: Parallel Computing for Heterogeneous Devices. Technical report, Khronos Group, 2009.
- [61] Khronos OpenCL Working Group. The OpenCL Specification. Technical report, 2009.
- [62] The Portland Group. PGI Fortran & C Accelerator Programming Model. Technical report, 2008.
- [63] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA Data Mining Software: An Update. *SIGKDD Explorations*, 11:10–18, 2009.
- [64] Mark Harris. *GPU Gems 2*, chapter 31: Mapping Computational Concepts to GPUs, pages 493–508. Addison-Wesley, 2005.
- [65] Vincent Heuveline and Jan-Philipp Weib. A Parallel Implementation of a Lattice Boltzmann Method on the ClearSpeed Advance™ Accelerator Board. Technical report, Karlsruhe Institute of Technology, 2007.
- [66] Brian Holland, Karthik Nagarajan, Chris Conger, Adam Jacobs, and Alan D George. RAT: A Methodology for Predicting Performance in Application Design Migration to FPGAs. In *High-Performance Reconfigurable Computing Technologies & Apps Workshop*, 2007.
- [67] Lee W. Howes, Paul Price, Oskar Mencer, Olav Beckmann, and Oliver Pell. Comparing FPGAs To Graphics Accelerators And The Playstation 2 Using A Unified Source Description. In *IEEE Conference on Field Programmable Logic and Applications*, pages 119–124, 2006.

- [68] J. R. Humphrey, D. K. Price, K. E. Spagnoli, A. L. Paolini, and E. J. Kelmelis. CULA: Hybrid GPU Accelerated Linear Algebra Routines. *SPIE Defense and Security Symposium (DSS)*, 2011.
- [69] Robin Hunter. *The Essence of Compilers*. Pearson Education, 1999.
- [70] AMD Inc. AMD OpenCL Examples. [Online] <http://developer.amd.com/GPU/ATISTREAMSDK>, [Accessed November 2009].
- [71] AMD Inc. AMD Stream Computing: Software Stack. Technical report, AMD Inc., 2007.
- [72] ATI Inc. ATI CTM Guide. Technical report, ATI Corporation, 2007.
- [73] ClearSpeed Inc. *The ClearSpeed Accelerated DFT Library*, 2006.
- [74] ClearSpeed Inc. *ClearSpeed Software Development Kit Reference Manual*, 2008.
- [75] ClearSpeed Inc. *The CSPX Accelerator Interface Library User Guide*, 2008.
- [76] John Kessenich, Dave Baldwin, and Rani Rost. The OpenGL Shading Language. Technical report, OpenGL, 2004.
- [77] Bill Kircos. An update on our graphics-related programs. [Online] Technology@Intel Blog <http://blogs.intel.com/technology/>, [Accessed August 2010].
- [78] Seth Koehler, John Currenri, and Alan D George. Performance Analysis Challenges and Framework for High Performance Reconfigurable Computing. *Parallel Computing*, 24:217–230, 2008.
- [79] David M Kunzman and Laxmikant V Kale. Towards a Framework for Abstracting Accelerators in parallel Applications: Experience with Cell. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, volume 54, pages 54:1–54:12, 2009.
- [80] Jakub Kurzak, Alfredo Buttari, Piotr Luszczek, and Jack Dongarra. The PlayStation 3 for High Performance Scientific Computing. Technical report, University of Tennessee Computer Science Technical Report, 2008.

- [81] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. In *Proceedings of Principles and Practice of Parallel Computing*, pages 101–110, 2009.
- [82] Aaron Eliot Lefohn. *Glift: Generic Data Structures for Graphics Hardware*. PhD thesis, University of California Davis, 2006.
- [83] Calle Lejdfors and Lennart Ohlsson. Implementing an Embedded GPU Language by Combining Translation and Generation. *Proceedings of the 2006 ACM symposium on Applied computins*, pages 1610–1614, 2006.
- [84] Erik Lindholm, Mark J Kilgard, and Henry Moreton. A User-Programmable Vertex Engine. *Proceedings of ACM SIGGRAPH*, pages 149–158, 2001.
- [85] William R Mark, R. Steven Glanville, Kurt Akely, and Mark J Kilgard. Cg: A System for Programming Graphics Hardware in a C-like Language. *ACM Transactions on Graphics*, 22:896–907, 2003.
- [86] Makoto Matsumoto and Takuji Nishimura. Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator. *ACM Transactions on Modeling and Computer Simulation*, 8:3–30, 1998.
- [87] Simon McIntosh-Smith. Practical Parallel Computing: Harnessing The Many-Core Future. [Online] <http://www.iee-cambridge.org.uk/arc/seminar07/slides/SimonMcIntoshSmith.pdf>, [Accessed January 2010]. Presentation given at The IET Cambridge Branch Seminar November 2007.
- [88] Simon McIntosh-Smith. Meeting the Performance Per Watt Power Challenge a 10X Increase in GFLOPS per Watt. [Presentation], 2004.
- [89] Oscar Mencer. ASC: A Stream Compiler for Computing with FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and System*, 25:1603–1617, 2006.
- [90] Nicholas Metropolis and S. Ulam. The Monte Carlo Method. *Journal of the American Statistical Association*, 44:335–341, 1949.

-
- [91] Mitronics. Low Power Hybrid Computing for Efficient Software Acceleration. Technical report, Mitronics AB, 2008.
- [92] Stefan Mohl. The Mitrion-C Programming Language. Technical report, Mitronics AB, 2006.
- [93] Nicholas Moore, Albert Conti, and Miriam Lesser. VForce: An Extensible Framework for Reconfigurable Computing. *IEEE Computer*, 40:39–49, 2007.
- [94] Kenneth Moreland and Edward Angel. The FFT on a GPU. *Proceedings Of The Acm Siggraph/Eurographics Conference On Graphics Hardware*, 1:112–119, 2003.
- [95] Walid A Najjar, Wim Bohm, Bruce A Draper, Jeff Hammes, Rober Rinker, J. Ross Beveridge, Minica Chawathe, and Charles Ross. High Level Language Abstraction for Reconfigurable Computing. *IEEE Computer*, 36:63–69, 2003.
- [96] Rajib Nath, Stanimire Tomov, and Jack Dongarra. Accelerating GPU kernels for dense linear algebra. *Proceedings of VECPAR 2010*, 6449:83–92, 2010.
- [97] Rajib Nath, Stanimire Tomov, and Jack Dongarra. An Improved MAGMA GEMM for Fermi GPUs. Technical report, University of Tennessee, 2010.
- [98] Yuri Nishikawa, Michihiro Koibuchi, Masato Yoshimi, Kenichi Miura, and Hideharu Amano. Performance Improvement Methodology for ClearSpeed CSX600. In *Proceedings of International Conference on Parallel Processing*, pages 77–85, 2007.
- [99] NVIDIA. CUDA Product Information Website. [Online], [Accessed February 2011].
- [100] NVIDIA. CUDA CUFFT Library. Technical report, NVIDIA, 2007.
- [101] NVIDIA. NVIDIA Tesla Computing Processor: Solve Tomorrows Computing Problems Today. Technical report, 2008.
- [102] NVIDIA. CUDA Best Practices Guide. Technical report, 2009.

- [103] NVIDIA. CUDA CUBLAS Library. Technical report, 2009.
- [104] NVIDIA. Cuda Occupancy Calculator Spreadsheet. NVIDIA SDK, 2009.
- [105] NVIDIA. OpenCL Overview. Technical report, NVIDIA, 2009.
- [106] NVIDIA. TESLA C2050 / C2070 GPU Computing Processor: Supercomputing at 1/10th the Cost. Technical report, 2010.
- [107] NVIDIA. CUDA Toolkit 3.2 Math Library Performance. [Presentation], February 2011.
- [108] Cyril Zeller NVIDIA. CUDA Tutorial. [Online] <http://people.maths.ox.ac.uk/~gilesm/hpc/NVIDIA/>, NVIDIA_CUDA_Tutorial_No_NDA_Apr08.pdf [Accessed August 2010].
- [109] Lars Nyland, Mark Harris, and Jan Prins. *GPU Gems 3*, chapter 31: Fast N-Body Simulation with CUDA, page 633673. Addison-Wesley, 2008.
- [110] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. GPU Computing. *Proceedings of the IEEE*, 96:879–899, 2008.
- [111] John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Kruger, Aaron E Lefohn, and Timothy J Purcell. A Survey of General Purpose Computation of Graphics Hardware. *Eurographics*, 26:80–113, 2005.
- [112] Peakstream. The Peakstream Platform:: High Productivity Software Development for Multi-Core Processors. Technical report, Peakstream, 2006.
- [113] Craig Peeper and Jason L Mitchell. Introduction to the DirectX 9 High Level Shading Language. Technical report, Microsoft, 2004.
- [114] Stefan Popov, Johannes Gnther, Hans-Peter Seidel, and Philipp Slusallek. Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Computer Graphics Forum*, 26:415–424, 2007.

- [115] S. A. Prah, M. Keijzer, S. L. Jacques, and A. J. Welch. A Monte Carlo Model of Light Propagation in Tissue. In *SPIE Proceedings of Dosimetry of Laser Radiation in Medicine and Biology*, pages 102–111, 1989.
- [116] William H Press, Brian P Flannery, Saul A Teukolsky, and William T Vetterling. *Numerical Recipes in C*. Cambridge, 1990.
- [117] William Pugh. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. *Communications of the ACM*, 8:102–114, 1992.
- [118] Timothy J Purcell, Ian Buck, William R Mark, and Pat Hanrahan. Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics (TOG)*, 21:703–712, 2002.
- [119] IBM Research. Innovation matters: The Cell architecture. [Online] <http://domino.research.ibm.com/comm/research.nsf/pages/r.arch.innovation.html>, [Accessed Dec 09].
- [120] Andrew Richards. The Codeplay Sieve C++ Parallel Programming System. Technical report, Codeplay, 2007.
- [121] M. W. Riley, J. D. Warnock, and D. F. Wendel. Cell Broadband Engine processor: Design and implementation. *IBM Journal of Research and Development*, 51:545–557, 2007.
- [122] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman³, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A Many-Core x86 Architecture for Visual Computing. Technical report, Intel Corporation, 2009.
- [123] Milan Sonka, Vaclav Hlavac, and Roger Boyle. *Image Processing, Analysis and Machine Vision*. Thompson Computer Press, 1996.
- [124] Sinmon Stegmaier, Magnus Strengert, Thomas Klein, and Thomas Ertl. A Simple and Flexible Volume Rendering Framework for Graphics-Hardware-based Raycasting. *Volume Graphics 2005 Eurographics/IEEE VGTC Workshop Proceedings*, 1:187–195, 2005.

- [125] Jone E Stone, David J Hardy, Ivan S Ufimtsev, and Klaus Schulten. GPU-accelerated molecular modeling coming of age. *Journal of Molecular Graphics and Modelling*, 29:116–125, 2010.
- [126] Magnus Strengert, Thomas Klein, Ralf Botchen, Simon Stegmaier, Min Chen, and Thomas Ertl. Spectral volume rendering using GPU based raycasting. *The Visual Computer*, 22:550–561, 2006.
- [127] Kevin Suffern. *Ray Tracking from the Ground Up*. A K Peters, Ltd., 2007.
- [128] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Pearson, 2006.
- [129] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses. *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 325–335, 2006.
- [130] Justin L Tripp, Maya B Gokhale, and Kristopher D Peterson. Trident: From High-Level Language to Hardware Circuitry. *IEEE Computer*, 40:28–37, 2007.
- [131] Sain-Zee Ueng, Melvin Lathara, Sara S. Baghsorkhi, and Wen mei W. Hwu. CUDA-Lite: Reducing GPU Programming Complexity. *Lecture Notes in Computer Science*, 5335:1–15, 2008.
- [132] Richard Wain, Ian Bush, Martyn Guest, Miles Deegan, Igor Kozin, and Christine Kitchen. An Overview of FPGA and FPGA Programming; Initial Experiences at Daresbury. Technical report, CCLRC, 2006.
- [133] Ian H Witten and Eibe Frank. *Data Mining : Practical machine learning tools and techniques*. Morgan Kaufmann, 2000.
- [134] Michael Wolfe. The PGI Accelerator Programming Model on NVIDIA GPUs. *PGI Insider*, 2009.
- [135] Fang Xu and Klaus Mueller. Accelerating Popular Tomographic Reconstruction Algorithms on Commodity PC Graphics Hardware. *IEEE Transactions on Nuclear Science*, 52:654–663, 2005.

Appendices

Appendix A

Porting Example: GEMM

A.1 Input Source

```
// does C= alphaAB + betaC
// C is of dimensions m*n
//A is of dimensions m*k
// B is of dimensions k*n
#include <stdlib.h>
#include <stdio.h>

int main() {

int m=2800;
int n=2800;
int k=2800;
int i;
float alpha=25.21;
float beta=42.52;
int x;
FILE * file;
float * a;
float * c;
float * b;
float *cOut;
```

```
a=(float*)malloc(sizeof(float)*m*k);
b=(float*)malloc(sizeof(float)*k*n);
c=(float*)malloc(sizeof(float)*n*m);
cOut=(float*)malloc(sizeof(float)*n*m);
file=fopen("data","rb");

//initialise bodies
for (i=0; i < k*m;i++) {
    char buf[999];
    fgets(buf,999,file);
    a[i]=atof(buf);
}

for (i=0; i < k*n;i++) {
    int x=i/n;
    int y=i - (x*n);
    char buf[999];
    fgets(buf,999,file);
    b[(y*k)+x]=atof(buf);
}

for (i=0; i < m*n;i++) {
    char buf[999];
    fgets(buf,999,file);
    c[i]=atof(buf);
}

fclose(file);

for (x=0; x < n*m;x++) {
    int y;
    int i= x/m;
    int j= x-(i*m);
```

```
// C is of dimensions m*n
//A is of dimensions m*k
// B is of dimensions k*n

    float sum=0.0;
    for ( y=0; y < m;y++) {
        sum+=a[ (i*k)+y]*b[(j*k)+y];
    }

    cOut[x]= (alpha*sum) + (beta*c[x]);

}

for (x=0; x < n*m;x++) {

    printf("[%d]=%f\n",x,cOut[x]);
}

}
```

A.2 Code Executing on CPU

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <sys/time.h>
#include <stdlib.h>
#include <stdio.h>
int main(){

int m = 2800;
int n = 2800;
int k = 2800;
int i;
float alpha = 25.21;
float beta = 42.52;
int x;
FILE *file;
float *a;
float *c;
float *b;
float *cOut;

a = (float *) (malloc(sizeof(float ) * m * k));
b = (float *) (malloc(sizeof(float ) * k * n));
c = (float *) (malloc(sizeof(float ) * n * m));
cOut = (float *) (malloc(sizeof(float ) * n * m));
file = fopen("data","rb");

for (i = 0;i < k * m;i++){
    char buf[999];
    fgets(buf,999,file);
    a[i] = atof(buf);
```



```
}

for (i = 0;i < k * n;i++){
    int x = i / n;
    int y = i - x * n;
    char buf[999];
    fgets(buf,999,file);
    b[y * k + x] = atof(buf);
}

for (i = 0;i < m * n;i++){
    char buf[999];
    fgets(buf,999,file);
    c[i] = atof(buf);
}

fclose(file);

#include "kernelloaders/kernel3.c"

for (x = 0;x < n * m;x++){
    printf("[%d]=%f\n",x,cOut[x]);
}

}
```

A.3 CUDA

A.3.1 Generated Host Code

```
int noDevicesKernel3;
cudaGetDeviceCount(&noDevicesKernel3);
if (noDevicesKernel3<1) {
printf("No Cuda Devices Found\n\r");
exit(1);
}
cudaSetDevice(0);
float *aKernel3Load;
cudaMalloc((void**)&aKernel3Load,(sizeof(float ) * m * k/sizeof(float))
           *sizeof(float));
cudaMemcpy(aKernel3Load,a,sizeof(float)*sizeof(float ) * m * k/sizeof(float)
           ,cudaMemcpyHostToDevice);
float *bKernel3Load;
cudaMalloc((void**)&bKernel3Load,(sizeof(float ) * k * n/sizeof(float))
           *sizeof(float));
cudaMemcpy(bKernel3Load,b,sizeof(float)*sizeof(float ) * k * n/sizeof(float)
           ,cudaMemcpyHostToDevice);
float *cKernel3Load;
cudaMalloc((void**)&cKernel3Load,(sizeof(float ) * n * m/sizeof(float))
           *sizeof(float));
cudaMemcpy(cKernel3Load,c,sizeof(float)*sizeof(float ) * n * m/sizeof(float)
           ,cudaMemcpyHostToDevice);
float *cOutKernel3Return;
cudaMalloc((void**)&cOutKernel3Return,(sizeof(float ) * n * m/sizeof(float))
           *sizeof(float));
int kernel3timesExecuted=0;
kernel3timesExecuted= (n*m - 0) / 1;

int kernel3alloc=ceil(sqrt(ceil((float)kernel3timesExecuted/(float)97)));
```

```

dim3 kernel3Grid(kernel3alloc,ceil(((double)kernel3timesExecuted
    /(((double)97*kernel3alloc))));
dim3 kernel3Block(97,1);
kernel3 <<<kernel3Grid,kernel3Block>>>(m,aKernel3Load,k,bKernel3Load,alpha
    ,beta,cKernel3Load,n,cOutKernel3Return,
    kernel3timesExecuted,kernel3alloc);
cudaMemcpy(cOut,cOutKernel3Return,sizeof(float)*(sizeof(float) * n *
    m/sizeof(float)),cudaMemcpyDeviceToHost);
cudaFree(aKernel3Load);
cudaFree(bKernel3Load);
cudaFree(cKernel3Load);
cudaFree(cOutKernel3Return);

```

A.3.2 Generated Device Code

```

__global__ void kernel3(int m,float *a,int k,float *b,float alpha,float beta,
    float *c,int n,float *cOutReturn,int kernel3timesExecuted,
    int kernel3alloc) {

int execNo= (((blockIdx.x*kernel3alloc)+blockIdx.y)*97)+threadIdx.x;
if ( execNo < kernel3timesExecuted){
    int x=(1*execNo)+0;
    int y;
    int i = x / m;
    int j = x - i * m;
    float sum = 0.0;
    for (y = 0; y < m; y++) {
        sum += a[i * k + y] * b[j * k + y];
    }
    cOutReturn[x] = alpha * sum + beta * c[x];
}
}

```

A.4 C_N

A.4.1 Generated Host Code

```

int noDevicesKernel3;
struct CSAPIState* kernel3State=NULL;
CSAPI_num_cards(&noDevicesKernel3);
if (noDevicesKernel3<1) {
printf("No Clearspeed Devices Found\n\r");
exit(1);
}
kernel3State=CSAPI_new();
CSAPI_connect(kernel3State,CSH_Private,CSC_Direct,"localhost",
              CSAPI_INSTANCE_ANY,0);
int noProcessorsKernel3;
int noPeKernel3;
CSAPI_num_processors(kernel3State,&noProcessorsKernel3);
CSAPI_num_pes(kernel3State,0,&noPeKernel3);
int kernel3timesExecuted=0;
kernel3timesExecuted=(n*m - 0) / 1;;

int noIterPerProcKernel3=ceil( (float)kernel3timesExecuted/
                              (float)noProcessorsKernel3);
int noIterPerPeKernel3=noIterPerProcKernel3/noPeKernel3;
struct CSAPIProcess *process3[noProcessorsKernel3];
CSAPIMemoryAddress cOutReturnKernel3[noProcessorsKernel3];
int writeMaxcOutKernel3[noProcessorsKernel3];
int writeMincOutKernel3[noProcessorsKernel3];
CSAPIMemoryAddress symbolAddraKernel3;
CSAPIMemoryAddress symbolAddrbKernel3;
CSAPIMemoryAddress symbolAddrcKernel3;
int kernel3MemPerProc=1048576000/noProcessorsKernel3;
int kernel3doubleload=0;

```

```

int procNo3;
for ( procNo3=0; procNo3 < noProcessorsKernel3; procNo3++) {
CSAPI_load(kernel3State,procNo3,"kernels/kernel3.csx",NULL,
          &(process3[procNo3]),CSAPI_NO_TIMEOUT);
}

for ( procNo3=0; procNo3 < noProcessorsKernel3; procNo3++) {
int kernel3totalLoad=0;
int loopCondxKernelMin3=procNo3 * noIterPerProcKernel3;
int loopCondxKernelMax3=loopCondxKernelMin3+noIterPerProcKernel3;
writeMincOutKernel3[procNo3]=sizeof(float ) * n * m/sizeof(float);
if (writeMincOutKernel3[procNo3]>loopCondxKernelMin3)
    writeMincOutKernel3[procNo3]=loopCondxKernelMin3;
writeMaxcOutKernel3[procNo3]=0;
if (writeMaxcOutKernel3[procNo3]<loopCondxKernelMax3)
    writeMaxcOutKernel3[procNo3]=loopCondxKernelMax3;
kernel3totalLoad+=(writeMaxcOutKernel3[procNo3]-writeMincOutKernel3[procNo3])
    *sizeof(float);
CSAPI_allocate_shared_memory(kernel3State,procNo3,CSM_Dram,
    (writeMaxcOutKernel3[procNo3]-writeMincOutKernel3[procNo3])
    *sizeof(float),sizeof(float),process3[procNo3],"cOutOut",
    &cOutReturnKernel3[procNo3]);
CSAPIMemoryAddress symbolAddrmKernel3;
CSAPI_get_symbol_value(kernel3State,process3[procNo3],"m",
    &symbolAddrmKernel3);
CSAPI_write_mono_memory(kernel3State,CSAPI_TRANSFER_PARAMS_SAFE,
    symbolAddrmKernel3,sizeof(int),&m);

kernel3totalLoad+=sizeof(float ) * m * k/sizeof(float)*sizeof(float);
int tmpProca=floor(kernel3totalLoad/kernel3MemPerProc);

if (kernel3doubleload==0) {
if (procNo3 == 0) {
CSAPI_allocate_shared_memory(kernel3State,tmpProca,CSM_Dram,

```

```

        (sizeof(float ) * m * k/sizeof(float))*sizeof(float),sizeof(float),
        process3[tmpProca], "a",&symbolAddrKernel3);
CSAPI_write_mono_memory(kernel3State,CSAPI_TRANSFER_PARAMS_SAFE,
        symbolAddrKernel3,(sizeof(float ) * m * k/sizeof(float))
        *sizeof(float),a);

}
if (tmpProca != procNo3) {
CSAPIMemoryAddress symTmp;
CSAPI_get_symbol_value(kernel3State,process3[procNo3], "a",&symTmp);
CSAPI_write_mono_memory(kernel3State,CSAPI_TRANSFER_PARAMS_SAFE,symTmp,
        sizeof(CSAPIMemoryAddress),&symbolAddrKernel3);
}
} else if (kernel3doubleload==1) {
CSAPI_allocate_shared_memory(kernel3State,procNo3,CSM_Dram,
        (sizeof(float ) * m * k/sizeof(float))*sizeof(float),sizeof(float),
        process3[procNo3], "a",&symbolAddrKernel3);
CSAPI_write_mono_memory(kernel3State,CSAPI_TRANSFER_PARAMS_SAFE,
        symbolAddrKernel3,(sizeof(float ) * m * k/sizeof(float))
        *sizeof(float),a);

}

CSAPIMemoryAddress symbolAddrKernel3;
CSAPI_get_symbol_value(kernel3State,process3[procNo3], "k",
        &symbolAddrKernel3);
CSAPI_write_mono_memory(kernel3State,CSAPI_TRANSFER_PARAMS_SAFE,
        symbolAddrKernel3,sizeof(int),&k);

kernel3totalLoad+=(sizeof(float ) * k * n/sizeof(float))*sizeof(float)
int tmpProcb=floor(kernel3totalLoad/kernel3MemPerProc);

if (kernel3doubleload==0) {
if (procNo3 == 0) {

```

```

CSAPI_allocate_shared_memory(kernel3State,tmpProcb,CSM_Dram,
    (sizeof(float ) * k * n/sizeof(float))*sizeof(float),sizeof(float),
    process3[tmpProcb],"b",&symbolAddrbKernel3);
CSAPI_write_mono_memory(kernel3State,CSAPI_TRANSFER_PARAMS_SAFE,
    symbolAddrbKernel3,(sizeof(float ) * k * n/sizeof(float))
    *sizeof(float),b);

}
if (tmpProca != procNo3) {
CSAPIMemoryAddress symTmp;
CSAPI_get_symbol_value(kernel3State,process3[procNo3],"b",&symTmp);
CSAPI_write_mono_memory(kernel3State,CSAPI_TRANSFER_PARAMS_SAFE,symTmp,
    sizeof(CSAPIMemoryAddress),&symbolAddrbKernel3);
}
} else if (kernel3doubleload==1) {
CSAPI_allocate_shared_memory(kernel3State,procNo3,CSM_Dram,
    (sizeof(float ) * k * n/sizeof(float))*sizeof(float),sizeof(float),
    process3[procNo3],"b",&symbolAddrbKernel3);
CSAPI_write_mono_memory(kernel3State,CSAPI_TRANSFER_PARAMS_SAFE,
    symbolAddrbKernel3,(sizeof(float ) * k * n/sizeof(float))
    *sizeof(float),b);

}

CSAPIMemoryAddress symbolAddralphaKernel3;
CSAPI_get_symbol_value(kernel3State,process3[procNo3],"alpha",
    &symbolAddralphaKernel3);
CSAPI_write_mono_memory(kernel3State,CSAPI_TRANSFER_PARAMS_SAFE,
    symbolAddralphaKernel3,sizeof(float),&alpha);
CSAPIMemoryAddress symbolAddrbetaKernel3;
CSAPI_get_symbol_value(kernel3State,process3[procNo3],"beta",
    &symbolAddrbetaKernel3);
CSAPI_write_mono_memory(kernel3State,CSAPI_TRANSFER_PARAMS_SAFE,

```

```

        symbolAddrbetaKernel3,sizeof(float),&beta);

kernel3totalLoad+=(sizeof(float ) * n * m/sizeof(float))*sizeof(float)
int tmpProcc=floor(kernel3totalLoad/kernel3MemPerProc);

if (kernel3doubleload==0) {
if (procNo3 == 0) {
CSAPI_allocate_shared_memory(kernel3State,tmpProcc,CSM_Dram,
        (sizeof(float ) * n * m/sizeof(float))*sizeof(float),sizeof(float),
        process3[tmpProcc],"c",&symbolAddrKernel3);
CSAPI_write_mono_memory(kernel3State,CSAPI_TRANSFER_PARAMS_SAFE,
        symbolAddrKernel3,
        (sizeof(float ) * n * m/sizeof(float))*sizeof(float),c);
}
if (tmpProca != procNo3) {
CSAPIMemoryAddress symTmp;
CSAPI_get_symbol_value(kernel3State,process3[procNo3],"c",&symTmp);
CSAPI_write_mono_memory(kernel3State,CSAPI_TRANSFER_PARAMS_SAFE,symTmp,
        sizeof(CSAPIMemoryAddress),&symbolAddrKernel3);

}
} else if (kernel3doubleload==1) {
CSAPI_allocate_shared_memory(kernel3State,procNo3,CSM_Dram,
        (sizeof(float ) * n * m/sizeof(float))*sizeof(float),sizeof(float),
        process3[procNo3],"c",&symbolAddrKernel3);
CSAPI_write_mono_memory(kernel3State,CSAPI_TRANSFER_PARAMS_SAFE,
        symbolAddrKernel3,
        (sizeof(float ) * n * m/sizeof(float))*sizeof(float),c);
}

CSAPIMemoryAddress symbolAddrnKernel3;
CSAPI_get_symbol_value(kernel3State,process3[procNo3],"n",

```



```

        &symbolAddrnKernel3);
CSAPI_write_mono_memory(kernel3State,CSAPI_TRANSFER_PARAMS_SAFE,
        symbolAddrnKernel3,sizeof(int),&n);

CSAPIMemoryAddress noExecAddr;
CSAPI_get_symbol_value(kernel3State,process3[procNo3],"noExec",&noExecAddr);
CSAPI_write_mono_memory(kernel3State,CSAPI_TRANSFER_PARAMS_SAFE,noExecAddr,
        sizeof(int),&noIterPerProcKernel3);
CSAPIMemoryAddress firstExecAddr;
int firstExec=procNo3*noIterPerProcKernel3;
CSAPI_get_symbol_value(kernel3State,process3[procNo3],"firstExec",
        &firstExecAddr);
CSAPI_write_mono_memory(kernel3State,CSAPI_TRANSFER_PARAMS_SAFE,firstExecAddr,
        sizeof(int),&firstExec);
CSAPI_run(kernel3State,process3[procNo3],NULL);
if (kernel3totalLoad < kernel3MemPerProc ) kernel3doubleload =1;

}
for ( procNo3=0; procNo3 < noProcessorsKernel3; procNo3++) {
CSAPI_wait_on_terminate(kernel3State,process3[procNo3],CSAPI_NO_TIMEOUT);
CSAPI_read_mono_memory(kernel3State,CSAPI_TRANSFER_PARAMS_SAFE,
        cOutReturnKernel3[procNo3], (writeMaxcOutKernel3[procNo3]
        -writeMincOutKernel3[procNo3])*sizeof(float),cOut
        +writeMincOutKernel3[procNo3]);

}
CSAPI_delete(kernel3State);

```

A.4.2 Generated Device Code: Non Buffered Kernel

```

#include <stdiop.h>
#include <stdlib.h>
#include <mathp.h>

```

```
#include <lib_ext.h>
int noExec;
int firstExec;
int noPerProc;
int m;
float* a;
int k;
float* b;
float alpha;
float beta;
float* c;
int n;
float* cOutOut;

int main(int argc, char**argv) {

int kernel3Loop;
poly int offset;
mono short SEMAPHORE=1;
mono short SEMAPHORE1=2;
sem_sig(SEMAPHORE);
noPerProc=ceil((float)noExec/96);
offset=(get_penum()*noPerProc)+firstExec;
for (kernel3Loop=0;kernel3Loop < noPerProc;kernel3Loop++) {

poly int x=(1*(offset+kernel3Loop))+0;
poly float tmpKernel0;
poly float tmpKernel1;
poly int y;
poly int i = x / m;
poly int j = x - i * m;
poly float sum = 0.0;
async_memcpy2p(SEMAPHORE1,&tmpKernel0,c+x,sizeof(float));
```

```

if (1<2) {
#include "kernel4.cn"
}
sem_wait(SEMAPHORE1);
tmpKernel1 = alpha * sum + beta * tmpKernel0;
sem_wait(SEMAPHORE);
async_memcpy2m(SEMAPHORE,cOutOut-firstExec+x,&tmpKernel1,sizeof(float));

}

sem_wait(SEMAPHORE);

}

```

A.4.3 Generated Device Code: Buffered Kernel

```

#define BUFFERSIZE 32

void process(poly float * mono inData, poly float * mono outData,poly int y,
            int bufferIter,poly float *sum){
    (* sum) += inData[(0*BUFFERSIZE)+bufferIter] *
              inData[(1*BUFFERSIZE)+bufferIter];
}

void loadData(mono short * SEMAPHORE, poly float * mono inData, int bufferNo,
            poly int i, poly int k,poly int j ) {
short size;
poly int offset=(bufferNo*BUFFERSIZE);
poly int y=offset;
size=BUFFERSIZE*sizeof(float);
async_memcpy2p(SEMAPHORE[0],inData+0*BUFFERSIZE,a+(i * k + y),size);

```

```
size=BUFFERSIZE*sizeof(float);
async_memcpy2p(SEMAPHORE[1],inData+1*BUFFERSIZE,b+(j * k + y),size);
}
```

```
void saveData(mono short * SEMAPHORE, poly float * mono outData,
             int bufferNo ) {
short size;
poly int offset=(bufferNo*BUFFERSIZE)+(get_penum()*noPerProc);
poly int y=offset;
}
```

```
void outBufferWait(mono short * SEMAPHORE) {
int i;
for (i=0; i < 0;i++) sem_wait(SEMAPHORE[i]);
}
```

```
void inBufferWait(mono short * SEMAPHORE) {
int i;
for (i=0; i < 2;i++) sem_wait(SEMAPHORE[i]);
}
```

```
int errorOffset;
mono short ISEMAPHORE[3][2];
mono short OSEMAPHORE[3][1];
poly float buffer1[2*BUFFERSIZE];
poly float buffer2[2*BUFFERSIZE];
poly float buffer3[2*BUFFERSIZE];
poly float buffer4[1];
poly float buffer5[1];
poly float buffer6[1];
poly float * mono inputBuffer[3];
```

```
poly float * mono outputBuffer[3];
short currentBuffer,bufferIter,innerLoopIter,bufferCount;
poly int bufferOffset;
noPerProc=ceil((float)noExec/96);
inputBuffer[0]=&buffer1;
inputBuffer[1]=&buffer2;
inputBuffer[2]=&buffer3;
outputBuffer[0]=&buffer4;
outputBuffer[1]=&buffer5;
outputBuffer[2]=&buffer6;

ISEMAPHORE[0][0]=3;
ISEMAPHORE[1][0]=4;
ISEMAPHORE[2][0]=5;
ISEMAPHORE[0][1]=6;
ISEMAPHORE[1][1]=7;
ISEMAPHORE[2][1]=8;

currentBuffer=0;
loadData(ISEMAPHORE[currentBuffer],inputBuffer[currentBuffer],0,i,k,j);
currentBuffer=1;
loadData(ISEMAPHORE[currentBuffer],inputBuffer[currentBuffer],1,i,k,j);
currentBuffer=2;
loadData(ISEMAPHORE[currentBuffer],inputBuffer[currentBuffer],2,i,k,j);

bufferCount=ceil(((float)m)/((float)BUFFERSIZE))-3;
bufferOffset=firstExec+(get_penum()*noPerProc);

for (bufferIter=0; bufferIter < bufferCount;bufferIter++) {

if (currentBuffer==0) {
    currentBuffer=1;
} else if(currentBuffer==1) {
    currentBuffer=2;
```

```
} else {
currentBuffer=0;
}

inBufferWait(ISEMAPHORE[currentBuffer]);

for (innerLoopIter=0; innerLoopIter< BUFFERSIZE;innerLoopIter++)
    process(inputBuffer[currentBuffer],outputBuffer[currentBuffer],
            bufferOffset+innerLoopIter,innerLoopIter,&sum);
loadData(ISEMAPHORE[currentBuffer],inputBuffer[currentBuffer],
        bufferIter+3,i,k,j);
outBufferWait(OSEMAPHORE[currentBuffer]);
saveData(OSEMAPHORE[currentBuffer],outputBuffer[currentBuffer],bufferIter);
bufferOffset+=BUFFERSIZE;
}

if (currentBuffer==0) {
currentBuffer=1;
} else if(currentBuffer==1) {
currentBuffer=2;
} else {
currentBuffer=0;
}

inBufferWait(ISEMAPHORE[currentBuffer]);
for (innerLoopIter=0; innerLoopIter< BUFFERSIZE;innerLoopIter++)
    process(inputBuffer[currentBuffer],outputBuffer[currentBuffer],
            bufferOffset+innerLoopIter,innerLoopIter,&sum);
outBufferWait(OSEMAPHORE[currentBuffer]);
saveData(OSEMAPHORE[currentBuffer],outputBuffer[currentBuffer],bufferCount);
bufferOffset+=BUFFERSIZE;

if (currentBuffer==0) {
currentBuffer=1;
```

```
} else if(currentBuffer==1) {
    currentBuffer=2;
} else {
currentBuffer=0;
}

inBufferWait(ISEMAPHORE[currentBuffer]);
for (innerLoopIter=0; innerLoopIter< BUFFERSIZE;innerLoopIter++)
    process(inputBuffer[currentBuffer],outputBuffer[currentBuffer],
        bufferOffset+innerLoopIter,innerLoopIter,&sum);

outBufferWait(OSEMAPHORE[currentBuffer]);
saveData(OSEMAPHORE[currentBuffer],outputBuffer[currentBuffer],bufferCount);
bufferOffset+=BUFFERSIZE;

if (currentBuffer==0) {
currentBuffer=1;
} else if(currentBuffer==1) {
    currentBuffer=2;
} else {
currentBuffer=0;
}

inBufferWait(ISEMAPHORE[currentBuffer]);
if (((bufferCount+3)*BUFFERSIZE) > m)
    errorOffset=m-((bufferCount+2)*BUFFERSIZE);
else
errorOffset=BUFFERSIZE;

for (innerLoopIter=0; innerLoopIter<errorOffset;innerLoopIter++)
    process(inputBuffer[currentBuffer],outputBuffer[currentBuffer],
        bufferOffset+innerLoopIter,innerLoopIter,&sum);
```

```
outBufferWait(OSEMAPHORE[currentBuffer]);
saveData(OSEMAPHORE[currentBuffer],outputBuffer[currentBuffer],bufferCount+1)

if (currentBuffer==0) {
currentBuffer=1;
} else if(currentBuffer==1) {
currentBuffer=2;
} else {
currentBuffer=0;
}

outBufferWait(OSEMAPHORE[currentBuffer]);
if (currentBuffer==0) {
currentBuffer=1;
} else if(currentBuffer==1) {
currentBuffer=2;
} else {
currentBuffer=0;
}

outBufferWait(OSEMAPHORE[currentBuffer]);
if (currentBuffer==0) {
currentBuffer=1;
} else if(currentBuffer==1) {
currentBuffer=2;
} else {
currentBuffer=0;
}

outBufferWait(OSEMAPHORE[currentBuffer]);
```


Appendix B

Kernel Description Metrics for FFT Application

App ID	Kernel ID	Problem Size	Intensity	Highest Precision	No Branch	Data Access	Data Write	No Iterations	Data Moved	Device
7	3	512	75395	DOUBLE	512	16509	3191	512	8388608	CPU
7	3	1024	168363	DOUBLE	1024	37002	5702	1024	33554432	CPU
7	3	2048	372179	DOUBLE	2048	82071	10374	2048	134217728	CPU
7	3	4096	815611	DOUBLE	4096	180388	19267	4096	536870912	GPU
7	3	8192	1774115	DOUBLE	8192	393393	36488	8192	2147483648	GPU
7	3	512	75395	FLOAT	512	16509	3191	512	4194304	CPU
7	3	1024	168363	FLOAT	1024	37002	5702	1024	16777216	CPU
7	3	2048	372179	FLOAT	2048	82071	10374	2048	67108864	CPU
7	3	4096	815611	FLOAT	4096	180388	19267	4096	268435456	GPU
7	3	8192	1774115	FLOAT	8192	393393	36488	8192	1073741824	GPU
7	6	512	1570	DOUBLE	0	253	127	9	8388608	CPU
7	6	1024	2228	DOUBLE	0	321	161	10	33554432	CPU
7	6	2048	3229	DOUBLE	0	396	198	11	134217728	CPU
7	6	4096	4836	DOUBLE	0	480	240	12	536870912	CPU
7	6	8192	7549	DOUBLE	0	572	286	13	2147483648	CPU
7	6	512	1570	FLOAT	0	253	127	9	4194304	CPU
7	6	1024	2228	FLOAT	0	321	161	10	16777216	CPU
7	6	2048	3229	FLOAT	0	396	198	11	67108864	CPU
7	6	4096	4836	FLOAT	0	480	240	12	268435456	CPU
7	6	8192	7549	FLOAT	0	572	286	13	1073741824	CPU

App ID	Kernel ID	Problem Size	Intensity	Highest Precision	No Branch	Data Access	Data Write	No Iterations	Data Moved	Device
7	8	512	35	DOUBLE	0	8	4	31	8388608	CPU
7	8	1024	35	DOUBLE	0	8	4	40	33554432	CPU
7	8	2048	35	DOUBLE	0	8	4	49	134217728	CPU
7	8	4096	35	DOUBLE	0	8	4	60	536870912	CPU
7	8	8192	35	DOUBLE	0	8	4	71	2147483648	CPU
7	8	512	35	FLOAT	0	8	4	31	4194304	CPU
7	8	1024	35	FLOAT	0	8	4	40	16777216	CPU
7	8	2048	35	FLOAT	0	8	4	49	67108864	CPU
7	8	4096	35	FLOAT	0	8	4	60	268435456	CPU
7	8	8192	35	FLOAT	0	8	4	71	1073741824	CPU
7	9	512	11	DOUBLE	0	2	2	262144	16777216	CPU
7	9	1024	11	DOUBLE	0	2	2	1048576	67108864	CPU
7	9	2048	11	DOUBLE	0	2	2	4194304	268435456	CPU
7	9	4096	11	DOUBLE	0	2	2	16777216	1073741824	GPU
7	9	8192	11	DOUBLE	0	2	2	67108864	4294967296	GPU
7	9	512	11	FLOAT	0	2	2	262144	8388608	CPU
7	9	1024	11	FLOAT	0	2	2	1048576	33554432	CPU
7	9	2048	11	FLOAT	0	2	2	4194304	134217728	CS
7	9	4096	11	FLOAT	0	2	2	16777216	536870912	GPU
7	9	8192	11	FLOAT	0	2	2	67108864	2147483648	GPU
7	10	512	75395	DOUBLE	512	16509	3191	512	8388608	CPU
7	10	1024	168363	DOUBLE	1024	37002	5702	1024	33554432	CPU
7	10	2048	372179	DOUBLE	2048	82071	10374	2048	134217728	CPU
7	10	4096	815611	DOUBLE	4096	180388	19267	4096	536870912	GPU
7	10	8192	1774115	DOUBLE	8192	393393	36488	8192	2147483648	GPU
7	10	512	75395	FLOAT	512	16509	3191	512	4194304	CPU
7	10	1024	168363	FLOAT	1024	37002	5702	1024	16777216	CPU
7	10	2048	372179	FLOAT	2048	82071	10374	2048	67108864	CPU
7	10	4096	815611	FLOAT	4096	180388	19267	4096	268435456	GPU
7	10	8192	1774115	FLOAT	8192	393393	36488	8192	1073741824	GPU

App ID	Kernel ID	Problem Size	Intensity	Highest Precision	No Branch	Data Access	Data Write	No Iterations	Data Moved	Device
7	13	512	1570	DOUBLE	0	253	127	9	8388608	CPU
7	13	1024	2228	DOUBLE	0	321	161	10	33554432	CPU
7	13	2048	3229	DOUBLE	0	396	198	11	134217728	CPU
7	13	4096	4836	DOUBLE	0	480	240	12	536870912	CPU
7	13	8192	7549	DOUBLE	0	572	286	13	2147483648	CPU
7	13	512	1570	FLOAT	0	253	127	9	4194304	CPU
7	13	1024	2228	FLOAT	0	321	161	10	16777216	CPU
7	13	2048	3229	FLOAT	0	396	198	11	67108864	CPU
7	13	4096	4836	FLOAT	0	480	240	12	268435456	CPU
7	13	8192	7549	FLOAT	0	572	286	13	1073741824	CPU
7	15	512	35	DOUBLE	0	8	4	31	8388608	CPU
7	15	1024	35	DOUBLE	0	8	4	40	33554432	CPU
7	15	2048	35	DOUBLE	0	8	4	49	134217728	CPU
7	15	4096	35	DOUBLE	0	8	4	60	536870912	CPU
7	15	8192	35	DOUBLE	0	8	4	71	2147483648	CPU
7	15	512	35	FLOAT	0	8	4	31	4194304	CPU
7	15	1024	35	FLOAT	0	8	4	40	16777216	CPU
7	15	2048	35	FLOAT	0	8	4	49	67108864	CPU
7	15	4096	35	FLOAT	0	8	4	60	268435456	CPU
7	15	8192	35	FLOAT	0	8	4	71	1073741824	CPU
7	17	512	40	DOUBLE	2	1	1	262144	8388608	CPU
7	17	1024	40	DOUBLE	2	1	1	1048576	33554432	CPU
7	17	2048	40	DOUBLE	2	1	1	4194304	134217728	CPU
7	17	4096	40	DOUBLE	2	1	1	16777216	536870912	CPU
7	17	8192	40	DOUBLE	2	1	1	67108864	2147483648	CPU
7	17	512	40	FLOAT	2	1	1	262144	4194304	CPU
7	17	1024	40	FLOAT	2	1	1	1048576	16777216	CPU
7	17	2048	40	FLOAT	2	1	1	4194304	67108864	CPU
7	17	4096	40	FLOAT	2	1	1	16777216	268435456	CPU
7	17	8192	40	FLOAT	2	1	1	67108864	1073741824	CPU