Applications Performance on GPGPUs

with the Fermi Architecture

Ahmed Mohamed Hassan Abdalla

August 1, 2011

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2011
Abstract

The latest GPU architecture released by Nvidia, code-named “Fermi”, is the most advanced computing GPU architecture ever built. Radical changes took place on the GPU computing architecture compared to Fermi’s predecessors such as the GT200 series and the G80s. In this dissertation the Fermi architecture is analysed, addressing the most prominent upgrades, by running extensive benchmark tests. These benchmarks included Nvidia’s own SDK package, the Scalable Heterogeneous Computing benchmark suite and the NAS Parallel Benchmark suit. All the benchmarks are described in detail and their results are analysed, and verify Fermi’s expected performance results.
Contents

Chapter 1 Introduction .............................................................................................................. 1

Chapter 2 Background and Literature Review ......................................................................... 2

2.1 GPUs before Fermi – G80 and GT200 .................................................................................. 2

2.2 Fermi – GF100/GT300 ........................................................................................................ 3

2.3 GPGPUs ............................................................................................................................... 5

2.3.1 CUDA Model ..................................................................................................................... 6

2.3.2 CUDA C ............................................................................................................................ 6

2.3.3 CUDA FORTRAN ............................................................................................................. 7

2.3.4 OpenCL ............................................................................................................................ 7

2.4 Applications ........................................................................................................................ 8

2.4.1 Nvidia’s Software Development Kit ................................................................................... 8

2.4.2 Scalable Heterogeneous Computing suite ......................................................................... 8

2.4.3 NAS Parallel Benchmarks ............................................................................................... 9

2.5 Related Work ....................................................................................................................... 9

2.6 Hardware Specs .................................................................................................................. 10

Chapter 3 CUDA SDK ............................................................................................................ 11

3.1 Test 1: Increased CUDA Cores ........................................................................................... 12

3.1.1 Scan – Background .......................................................................................................... 12

3.1.2 Scan – Results and Analysis ............................................................................................ 13

3.2 Test 2: A True Cache Hierarchy ........................................................................................... 14

3.2.1 Transpose - Background .................................................................................................. 14

3.2.2 Transpose - Results and Analysis .................................................................................... 15
Acknowledgements

I am very grateful to Ms. Xu Guo for her very advice and supervision and extreme patience during this dissertation. I would also like to thank the EPCC department for providing me with access to state-of-the-art technologies. I would like to also thank God and my parents who gave me this life changing experience. Finally I would like to thank my friend Ahmadi who has been with me all the way, pushing me to move forward, giving me advice and making me believe in myself.
Chapter 1

Introduction

The discussion about what Graphical Processing Units (GPUs) are, and how beneficial they could be when porting scientific applications for acceleration on GPUs, is a topic that has been exhausted. GPUs have imprinted their existence in the HPC arena, since 2007 when the first version of CUDA was released [1].

Another huge step was taken by Nvidia when they released Fermi to the market. The improved features Fermi has over its GPU predecessors are numerous and should be thoroughly analysed and the results documented. The improved features include but are not limited to, Error Correction Codes (ECC) that are not so beneficial to the graphical environment, but are critical for financial and mission critical tasks that work for extended amount of times, and fear losing precision from the surrounding environment that could cause soft errors. Also, a true cache hierarchy has been setup on Fermi, including 64KB configurable L1 caches. The notion of maximum parallelism is one step closer, as Fermi has implemented concurrent kernels i.e. kernel launches can now be issued in parallel. Double precision (DP) floating point arithmetic is another feature that has seen astonishing upgrades, now being eight times faster than Fermi’s predecessor GPUs, the GT200 series, and is now precisely half as fast as single-precision math operation [2]. Many other improvements were made and are going to be addressed and analysed in this dissertation.

The background and literature review for this project is discussed in Chapter 2, where the architecture of Fermi and its previous predecessors is explained. The CUDA model is also explained in Chapter 2 alongside with the programming frameworks available today, such as CUDA C, CUDA FORTRAN and OpenCL. A short summary on the benchmarks used is also given. The chapter concludes by stating some previous related work. Chapter 3, 4 and 5 discuss the applications chosen for benchmarking on Fermi, and reports the analysis of how well they perform on both, Fermi and its GPU predecessors. Those chapters start by giving some background on the benchmarks used, and then moves on to the analysis of running those benchmarks. Chapter 6 states the conclusion, and discusses possible future work related ideas.
Chapter 2

Background and Literature Review

In this chapter, the background and literature review for the dissertation will be given. The project aims at evaluating Fermi’s architecture by running several benchmarking and scientific applications, including Nvidia’s SDK, the SHOC and the NPB. The architecture of both, Fermi’s GPU predecessors and Fermi is going to be discoursed. A summarised discussion of the choice of applications to be run on Fermi will also be presented followed by related work.

2.1 GPUs before Fermi – G80 and GT200

![Figure 1 G80 Architecture][3]
The G80 cards, first released in 2006, are made of 681 million transistors [3]. It was the first GPU architecture released that was based on the unified shader. A unified shader based architecture is where all operations are performed through scalar unified shader pipelines also known as streaming processors (SPs). SPs consist of a floating point unit, integer unit and an ALU. G80 has 128 SPs grouped into 8 Symmetrical Multiprocessors (SMs). Every 2 SMs lies in one Texture Processing Unit (TPC) Figure 1. In other words, G80 had 128 SPs inside 16 SMs that was inside 8 TPCs. Every SM contains 16Kb shared memory, some read only L1 cache memory represented as 8 Kb constant memory - 64Kb in total, and 16Kb texture memory - 128Kb in total. In addition, every SM has got a local warp scheduler, dispatch unit, register unit, eight texture filtering units and four texture load/store units used for operations such as sin, cos, and exp. It is also used for DP floating point operations.

![Figure 2 G200 Architecture](image)

Though the number of transistors has changed from 681 million to 1.4 billion [4], the GT200 cards did not have any major significant improvements over its predecessor, the G80. As apparent from Figure 2, the total number of SPs became 240 instead of 128 on G80. Each TPC holds now 3 SMs. The new formation became, 240 SPs inside 30 SMs, 24 SPs each, inside 10 TPCs. The GT20 has 8 texture load/store units per TPC. The L1 read only texture cache has got a size increase from 16Kb to 24Kb per TPC becoming 240Kb in total. Though the number of texture filtering units didn't change between the GT200 and the G80, their performance was improved significantly [5].

2.2 Fermi – GF100/GT300

Also known as GT300 or GF100, the Fermi GPU computing architecture is the latest Nvidia release to the market. The previous GPU architecture has been restructured from top to down, and significant improvements have been added to both the software and the hardware side of Fermi. The first Fermi released GPU, featured 3.0 billion transistors [6], and 512 CUDA cores (512 SPs). This is higher than double the amount of SPs available in the GT200, Fermi’s predecessor. The latest formation became, 512 SPs inside 16 SMs, 32 SPs each, inside 16 Graphical Processing Clusters (GPCs). The Nvidia Giga-Thread engine, Figure 3, assigns, maintains and controls thread block scheduling across all the SMs.
Figure 3 GF100 Architecture [7]

Figure 4 A Graphical Processing Cluster (GPC) on Fermi [6]
Fermi GPUs have three main components; Graphics Processing Clusters (4 GPCs), Streaming Multiprocessors (SMs), and memory controllers. Every SP has an ALU and a floating point unit (FPU), both fully pipelined, and each FPU has the ability of producing IEEE compliant DP results in two clock cycles [7]. Each SM has now 64KB of local SRAM (shared memory) linked to it, compared to 16KB on the GT200 [8]. The reason behind calling it now the “local SRAM” is that it has become a 64KB configurable L1 cache hierarchy, with 16KB reserved for shared memory and 48KB of cache space or the other way round. Unlike G80 and GT200, the L1 caches function like real CPU caches and are not read only. L2 caches were added as a backup to the L1 cache, and are sized 768KB. As it can be seen in Figure 3, the L2 cache is connected to all the SMs on the card. Both L1 and L2 caches support write-back and write-through to global memory [7]. Additionally, the entire memory hierarchy is ECC protected, making Fermi the first GPU computing architecture to incorporate such a capability [9].

<table>
<thead>
<tr>
<th>GPU</th>
<th>G80</th>
<th>GT200</th>
<th>Fermi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transistors</td>
<td>681 million</td>
<td>1.4 billion</td>
<td>3.0 billion</td>
</tr>
<tr>
<td>CUDA Cores</td>
<td>128</td>
<td>240</td>
<td>512</td>
</tr>
<tr>
<td>Double Precision Floating Point Capability</td>
<td>None</td>
<td>30 FMA ops / clock</td>
<td>256 FMA ops / clock</td>
</tr>
<tr>
<td>Single Precision Floating Point Capability</td>
<td>128 MAD ops/clock</td>
<td>240 MAD ops / clock</td>
<td>512 FMA ops /clock</td>
</tr>
<tr>
<td>Warp schedulers (per SM)</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Special Function Units (SFUs) / SM</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Shared Memory (per SM)</td>
<td>16 KB</td>
<td>16 KB</td>
<td>Configurable 48 KB or 16 KB</td>
</tr>
<tr>
<td>L1 Cache (per SM)</td>
<td>None</td>
<td>None</td>
<td>Configurable 16 KB or 48 KB</td>
</tr>
<tr>
<td>L2 Cache (per SM)</td>
<td>None</td>
<td>None</td>
<td>768 KB</td>
</tr>
<tr>
<td>ECC Memory Support</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Concurrent Kernels</td>
<td>No</td>
<td>No</td>
<td>Up to 16</td>
</tr>
<tr>
<td>Load/Store Address Width</td>
<td>32-bit</td>
<td>32-bit</td>
<td>64-bit</td>
</tr>
</tbody>
</table>

Table 1 GPU Architecture Comparisons - 1 [10]

2.3 GPGPUs

General Purpose Computation on Graphical Processing Units (GPGPUs) is an approach used to access GPUs as if they were CPUs and use them for acceleration of non-graphical applications [11]. The GPU can no longer just be used for graphics purposes. It is a massively parallel stream processor with 32 bit floating point support, 64 bit on Fermi, has a flexible programming model with the introduction of frameworks like CUDA, OpenCL and so on, and has a high memory bandwidth. Major scientific research has been done in the area of GPGPUs, and numerous results have been
released on how well GPUs perform in comparison with their counter-fit CPUs. GPUs are processors that were built inherently to deal with graphical operations. What that meant is that all threads dealt simultaneously with an image pixel on an individual level. This massive power to work in parallel is a basic need that could benefit many scientific applications, which take prolonged amount of time to accomplish their tasks or produce significant results. Certainly not all applications would fit best for practising GPGPUs. Ideal applications to target GPGPU have large data sets, require high level of parallelism, have high arithmetic/computational intensity, need minimal CPU intervention to complete their tasks and most importantly, have minimal dependencies between the data elements.

2.3.1 CUDA Model

CUDA stands for Compute Unified Device Architecture. It is a parallel development framework model manufactured by Nvidia to be explicitly used on their released GPU cards. It assists programmers to port their applications on to GPUs for acceleration, and hides from them intricate tasks that involve dealing with complex hardware operations that were present in programs like OpenGL and Microsoft 3D. The CUDA Model is fairly straightforward. The entity that is accessed by all the threads of the GPU is called the Kernel. A single call is made to the kernel, and then all the threads start the kernel in parallel. A kernel is launched in the following manner:

\[ \text{kernel} \leftarrow \text{grid\_dimension, block\_dimension} \rightarrow \text{(parameters)} \]

The block dimension in the kernel launch shown above represents the number of threads that are called within each block. The grid dimension specifies the number of blocks that are called within each grid. See Figure 5. Each thread called within a block has access to the shared memory in the matching SM. Threads within an block have access to the same shared memory, can communicate with each other and have synchronisation barriers with each other, but not with threads from other blocks. The only way a thread within a block can communicate with another thread in another block is through the global memory. All threads have access to global, texture and constant memory. Threads are grouped into blocks, and blocks are grouped into grids. A block gets scheduled to execute on any SM and threads within that block get to share the shared memory and registers of that SM. Threads within a block are subdivided into individual warps of 32 threads each. Inside an SM, one or more warps get processed in parallel [12].

2.3.2 CUDA C

In June 2007, Nvidia released its first version of CUDA [13]. It is an extension to the C/C++ languages. A typical CUDA C code defines a kernel like any other C function, but adds the __global__ identifier to it. Other functions called could involve functions that are used to copy to and from the host’s memory to the device memory or vice versa, allocate and de-allocate memory on host or device. Host would signify a CPU and device would mean a GPU.
The CUDA C files are of type “.cu” and are compiled with the nvcc compiler. The nvcc compiler ships with the CUDA toolkit that has all the development tools, libraries, and documentation that are necessary for creating applications for the CUDA architecture. CUDA has been renamed to CUDA C, when CUDA FORTRAN was released.

![CUDA Model Layout](image)

**Figure 5 CUDA Model Layout [12]**

### 2.3.3 CUDA FORTRAN

Later along the years, Nvidia has established the need to present its programmable GPU to a wider selection of programmer audience, so has started working with the PGI group for an effort to address that requirement. In 2010, Nvidia and PGI released CUDA FORTRAN which is supported on Linux, Mac OS X and Windows [14]. It provides FORTRAN functionality to write kernels that can be executed on the device, copy data to and from host and device, allocating and de-allocating memory on device and is very similar to CUDA C in that it delivers similar functionality.

### 2.3.4 OpenCL

Unlike CUDA C and CUDA FORTRAN that function only on Nvidia’s GPUs, Open Computing Language (OpenCL) is intended for a wider selection or multi-core architectures. OpenCL was developed by the Khronous group with the participation of industry leading companies to address programming on heterogeneous architectures that incorporate multi-core CPU’s, GPUs and other types of processors. OpenCL, a subset of the C programming language, was released in December 2008 and now is an industry standard for programming on heterogeneous systems. It provides parallel computing abilities through task-based and data-based parallelism [15].
2.4 Applications

In order to analyse the Fermi architecture thoroughly, a set of tests and benchmark suits had to be chosen carefully, in order to evaluate most if not all the upgrades that took place on Fermi. Fermi’s most improved features include a dramatic increase in the number of CUDA cores, a true cache hierarchy, the Giga-thread engine, a substantial boost to the DP floating operation performance and increased precision, a PCIe Gen 2.0 high speed data transfer and ECC support [16]. Analysing these improvements, and comparing their results to results of previous GPU predecessors, would turn out to be a daunting task if the applications tested on Fermi were not coherent or were scattered from all over areas and fields. Standard benchmark suits were the only way that the results produced would be assured correctness and reliably, because comparisons could be made to exiting documented results done in the past or elsewhere. Benchmarks that addressed such unique requirements included Nvidia’s SDK, the SHOC and the NPB.

2.4.1 Nvidia’s Software Development Kit

Nvidia’s SDK Code is a complete set of libraries and illustrative codes written in CUDA C/C++/OpenCL that cover chief physics and chemistry simulations written mainly to operate on Nvidia’s GPUs [17]. The codes are fairly less complex and can be used to compare performance differentiation between Fermi and earlier GPUs, without the necessity of any code porting. Some complex code samples inside the SDK bundle have white papers that describe in detail the mechanics of the code provided. Few codes of the SDK bundle have been selected to test major Fermi features. The SDK code is obtainable online free of charge and is exists under Nvidia’s licensing agreement [18].

2.4.2 Scalable Heterogeneous Computing suite

The Scalable Heterogeneous Computing benchmark suite (SHOC) written by A. Danalis et al [19] are a couple of benchmark codes written in both OpenCL and CUDAC for stress, performance and comparison purposes on heterogeneous architectures including graphical processing units, multi-core processors and hybrid architectures. The suit is compromised of codes with varying complexities ranging from low-level measurement s of data bus transfers, to complex industrial codes such as the S3D turbulent combustion simulation. The range of code complexities in the suite makes it a perfect runner for benchmarking Fermi. The code is also obtainable online free of charge and exists under the Modified-BSD licensing agreement available in the downloadable package [20].
2.4.3 NAS Parallel Benchmarks

The NAS Parallel Benchmarks (NPB) released by the NASA Advanced Supercomputing division (NAS) [21] is a chain of computational fluid dynamics (CFD) applications that are developed in different programming flavours (C/C++/Fortran). They were designed primarily to examine performance on large parallel super computers. The benchmarks have also been ported to frameworks such as Message Passing (MPI) and Shared Memory Programming (OpenMP) which indicates that the codes do lend themselves to being parallelised. In order to get a copy of the NPB code, registration on the NAS website is obligatory, and the code exists under the NASA Open Source Initiative licensing contract [22].

2.5 Related Work

Extensive research has been done evaluating the newly released Fermi architecture. One of the most recent papers that has a similar track [23], proposed and evaluated two parallel implementations of Multi-dimensional Ensemble Empirical Mode Decomposition for multi-core CPUs and many-core GPUs. On the Tesla C2050, the double precision implementation showed a speed up of 48.6X in comparison with the sequential C version. The multi-core OpenMP implementation showed a speed up of 11.3X on a two Intel Xeon x7550 CPUs. Interestingly, the paper had experimental data results that showed that the CUDA implementation developed will permit even more continuous reduction in execution time, in regards to the future generation of GPUs through increased parallelism.

Another research [24] analyses the performance of the OP2 “active” library. Runtime performance results are presented in the paper for a demonstrative unstructured mesh application developed using OP2 on a variation of many-core processor systems including Intel Xeon Penryn and current Nehale micro-architectures and GPUs including Nvidia GTX260 and Tesla C2050. OpenMP and CUDA parallel versions are used and compared on an industrial sized mesh consisting of about 1.5 million edges. The performance results on the Tesla C2050 show that for single precision the speedup was 3.5X and for double precision a speedup of 2.3X was obtained, when compared with the Intel multi-core processors executing up to 16 OpenMP threads. The results in the paper suggested reasonable performance boosts by the GPUs for this class of applications, but emphasized very important concerns with the GPUs used, such as memory bandwidth limitations on multi-core architectures at rising scale, which can hinder and limit the attained performance.
2.6 Hardware Specs

An Nvidia’s GPU Compute Capability stands for the capability of the device architecture being addressed, characterized by its major number and minor number. The major number stands for the GPU core architecture in question and the minor number stands for the incremental changes that were added to that architecture. A device that has a compute capability higher than or equal to 2.0 is a Fermi architecture.

The three systems used for all the tests run in this dissertation are GeForce 8400GS, Tesla M1060 and Tesla C2050. The 8400GS belongs to the GT80 series, the M1060 belongs to the GT200 series and C2050 is Fermi/GF100 architecture. The M1060 lies under the NESS server machine, while C2050 and 8400GS lie under the Fermi-1 server machine, see Table 2. In all the discussion to follow, this is the way they will be addressed, without mentioning “GeForce” or “Tesla”. Also whenever C2050 is used, it implies Fermi.

<table>
<thead>
<tr>
<th>Specification</th>
<th>Ness</th>
<th>Fermi-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel(R) Xeon® CPU E5504 @ 2.00GHz</td>
<td>Intel(R) Xeon(R) CPU X5650 @ 2.07GHz</td>
</tr>
<tr>
<td>GPU</td>
<td>Nvidia Tesla M1060</td>
<td>Nvidia Tesla C2050</td>
</tr>
</tbody>
</table>

Table 2 Server Machines

Table 3 is a table that shows the main features of the three GPU architectures used. The data displayed was produced by running the deviceQuery code available in Nvidia’s SDK package suite.

<table>
<thead>
<tr>
<th></th>
<th>8400GS</th>
<th>M1060</th>
<th>C2050</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Capability</td>
<td>1.1</td>
<td>1.3</td>
<td>2.0</td>
</tr>
<tr>
<td>Total amount of global memory</td>
<td>255MB</td>
<td>4GB</td>
<td>3GB</td>
</tr>
<tr>
<td>CUDA Cores</td>
<td>8</td>
<td>240</td>
<td>448</td>
</tr>
<tr>
<td>Constant Memory</td>
<td>64KB</td>
<td>64KB</td>
<td>64KB</td>
</tr>
<tr>
<td>Shared Memory per block</td>
<td>16KB</td>
<td>16KB</td>
<td>48KB</td>
</tr>
<tr>
<td>Registers per block</td>
<td>8192</td>
<td>16384</td>
<td>32768</td>
</tr>
<tr>
<td>Warp size</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td>512</td>
<td>512</td>
<td>1024</td>
</tr>
<tr>
<td>Maximum sizes of each dimension of a block</td>
<td>512 x 512 x 64</td>
<td>512 x 512 x 64</td>
<td>1024 x 1024 x 64</td>
</tr>
<tr>
<td>Maximum sizes of each dimension of a grid</td>
<td>65535 x 65535 x 1</td>
<td>65535 x 65535 x 1</td>
<td>65535 x 65535 x 65535</td>
</tr>
<tr>
<td>Maximum memory pitch</td>
<td>2GB</td>
<td>2GB</td>
<td>2GB</td>
</tr>
<tr>
<td>Texture alignment</td>
<td>256B</td>
<td>256B</td>
<td>512B</td>
</tr>
<tr>
<td>Clock rate</td>
<td>1.40 GHz</td>
<td>1.30 GHz</td>
<td>1.15 GHz</td>
</tr>
<tr>
<td>Concurrent copy and execution</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Run time limit on kernels</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Support host page-locked memory mapping</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Concurrent kernel execution</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Device has ECC support enabled</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 3 GPU Architecture Comparisons - 2

10
Chapter 3

CUDA SDK

The Nvidia’s CUDA Software Development Kit (SDK) [25] is a set of examples with source code, utilities, and white papers to help programmers write applications with CUDA. It could also be used as benchmarks that measure performance and evaluate upgrades across different and developing GPU architectures. It includes dozens of code samples covering a wide range of applications ranging from simple techniques such as code integration and efficient loading of custom data types to complex libraries and real world applications such as image convolution, Black-Scholes options pricing and binomial options pricing. And most importantly it includes extensive performance bandwidth tests that are going to be discussed in later sections.

Most of the SDK example codes were run on the three GPUs, 8400GS, M1060 and C2050. This generated a huge amount of data that could have not been included and analysed in this dissertation. A decision had to be made on what applications to include and which to exclude. Finally, four SDK example applications were chosen and analysed. The main reason for choosing those applications was that they addressed the most newly advanced features of the Fermi architecture compared to its previous GPU predecessors. The following sections are going to discuss those SDK example codes as tests, give a background on each and then separately analyse each one of them. By no means do the discussed tests address each and every improvement made on the Fermi architecture, but attend the most prominent ones.
3.1 Test 1: Increased CUDA Cores

3.1.1 Scan – Background

The most notable improvement of the Fermi architecture compared to previous GPU architectures is the increased number of CUDA cores. At the least, this certainly means a higher cumulative GPU throughput for applications that are intensive in computations, but require less or no memory access. As mentioned in [26], single precision calculations reach over a Teraflop on a Fermi GPU.

A hungry mathematical function is the scan operation. It is a simple and a common parallel algorithm building block also known as the prefix-sum procedure. Given an array of numbers, scan computes a new array in which each element is the sum of all the elements before it in the input array. This operation is computation intensive and scales up to $O(n)$ when run on a CPU sequentially.

$$[b_0, b_1, ..., b_{n-1}],$$

prefix-sum returns

$$[X, b_0, (b_0 \Theta b_1), ..., (b_0 \Theta b_1 \Theta ... \Theta b_{n-2})]$$

where $\Theta$ is binary associative operator with identity $X$, and an array of $n$ elements

The Scan code example available in the SDK package demonstrates an efficient CUDA implementation of parallel prefix sum. The CUDA code addresses different implementations of the scan algorithm ranging from naive ones with complexity of $O(n \cdot \log_2(n))$ operations to implementations that are optimised for the use of the built in GPU shared memory and registers to for fast cache memory access that require only $O(n)$ operations, as such becoming at the least $n$ times faster than a sequential run on a single-threaded CPU.

The CUDA C Scan code uses the idea of balanced binary trees on the input buffer where the prefix sum is calculated by brushing from the top of the tree to its root or vice versa. The input array is copied to the shared memory, where the scan operations are applied. Two main functions are used; the brush-up stage and the brush-down stage. In brush-up, the tree is traversed from the bottom to the root computing partial sums at internal nodes of the tree. Parallel reduction also takes place at the end of this stage, where the root node has the sum of all nodes in the buffer. In brush-down, the tree is traversed from the root all the way down to the bottom, using the partial sums from the brush-up phase to build the scan in place on the buffer. Refer to [27] for more details on the CUDA scan operation.
3.1.2 Scan – Results and Analysis

FLOPS stands for floating point operations per second and is the measure of a computer's performance. There are many factors that have to be taken into account when measuring computer performance other than raw floating-point computing speed, such as I/O performance, inter-processor communication, cache coherence, and the memory hierarchy. In addition, following Amdahl’s law that states that the speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program, one can conclude that just measuring throughput, the amount of data processed in a given period of time, of a processor one cannot get an accurate measure of how that processor performs. However, a processor’s throughput is one of the very significant factors that affect computer performance.

The Scan CUDA code performs the prefix-sum operations on integer arrays ranging from 4 elements to 262144 elements. The throughput on each machine for different sizes is displayed in Table 4.

<table>
<thead>
<tr>
<th>Array Size - Short (Elements)</th>
<th>8400GS</th>
<th>M1060</th>
<th>C2050</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (s)</td>
<td>0.01822</td>
<td>0.00290</td>
<td>0.00049</td>
</tr>
<tr>
<td>Throughput (MegaElements/s)</td>
<td>0.0562</td>
<td>0.3526</td>
<td>2.0954</td>
</tr>
<tr>
<td>Array Size - Large (Elements)</td>
<td>262144</td>
<td>262144</td>
<td>262144</td>
</tr>
<tr>
<td>Time (s)</td>
<td>0.03533</td>
<td>0.00816</td>
<td>0.00103</td>
</tr>
<tr>
<td>Throughput (MegaElements/s)</td>
<td>7.4199</td>
<td>32.1245</td>
<td>253.4506</td>
</tr>
</tbody>
</table>

Table 4 Scan throughput of the GPUs on different array sizes

The 8400GS has 8 CUDA cores at 1.40 GHz each, followed by M1060 which has 240 CUDA cores at 1.30 GHz each, followed by C2050 which has 448 CUDA cores with 1.15 GHz each. Taking into account the mentioned attributes of each architecture, one can deduce safely that just by increasing the number of processing GPU cores, the throughput of any application should increase. Certainly, if the application included many memory type transfers, that theory may not apply, but in this Scan CUDA SDK example all the code does is copying the input buffer to the built in shared memory and processing it there. Other factors that may have played a role in the increased throughput could account to Fermi’s new capabilities like the Parallel data cache line, but since the fact of increased cores is being the aspect tested here, Fermi’s throughput has certainly transferred GPUs to a whole new level. Running the code through the compute profiler, M1060 had 4 active blocks per SM with 1024 threads, while C2050 had 6 blocks per SM with 1536 threads at any given time, meaning more active SMs and so the higher number of active cores.
3.2 Test 2: A True Cache Hierarchy

3.2.1 Transpose - Background

The new Fermi architecture supports a true cache hierarchy, L1/L2 cache structure, with on-chip shared memory resulting in incomparable throughput and accelerating capabilities. Refer to Section 2.2 for more details on the new cache hierarchy.

One of the major headaches of using CUDA on GPUs for software acceleration is that a limited amount of register space was allowed for each thread before they started overflowing to memory off chip, which can cost hundreds of clock cycles wasted, degrading application performance [28]. Rather than increasing the size of the registers for each SM on the new Fermi architecture, Nvidia chose fittingly to add an actual true cache hierarchy structure. Inevitably, when threads require more registers than the hardware can provide, they will first spill into L1 cache. Just like in CPUs the L1 cache is so much faster than the L2 cache and the L2 cache is so much faster than fetching data from main global memory. If L1 cache is full, or suffers other conflicts, these registers will leak into L2 cache, which is significantly larger than L1 cache. In other words, programs can now be written without the need to worry on register spilling. The negative impact will still be there, but not nearly as much as it did in the previous generations of GPU architectures before Fermi.

The Matrix Transpose SDK code example relies heavily on the use of different types of memory structures including registers, shared memory, L1 and L2 caches if available. The code is written to optimise a matrix transpose of floats, addressing eight different GPU kernels, each of which transposes a matrix using a specific method and uses a certain optimisation technique [29]. Those kernels progressively improve performance through different GPU optimisation techniques such as coalescing, removing shared memory bank conflicts, and eradicating partition camping.

For each kernel call, there is outer loop timing and inner loop timing. The outer loop timing takes place in the main program, when the kernel launch is called inside a loop. The inner loop timing takes place inside each kernel where the transpose operation happens inside a loop. The two methods of timing are beneficial; the outer loop timing is used as an overall performance metric, and the inner loop timing compares data movement times within a kernel.

The copy kernel just copies the matrix of the input buffer to the output buffer. The first kernel is the naïve transpose that allows each CUDA thread executing the kernel to transpose four elements from one column of the input matrix to one row of the output matrix, with no optimisation techniques at all. The kernels that follow address issues such as coalesced global memory accesses, shared memory bank conflicts and removing the padding of the shared memory arrays to improve performance.
3.2.2 Transpose - Results and Analysis

The most effective method available for analysing an application that makes heavy use of different types of GPU memory is the CUDA Compute Visual Profiler [30]. The Matrix Transpose SDK example was profiled using Nvidia’s CUDA visual profiler version 4.0.7 [31]. The main area of interest in this session, was observing how the caches were used on each architecture. As such the “Instructions” profiler counters were unmarked;

![Image](https://example.com/image.png)

**Figure 6 CUDA Visual Profiler; Session Profiler Counters**

Without the need of any further analysis, it is clear from *Figure 7*, the advancements made on the cache hierarchy on the Fermi architecture;

![Image](https://example.com/image.png)

**Figure 7 CUDA Visual Profiler; Cache structure on different GPU architectures**
In *Figure 7*, both 8400GS and M1060 display a cache structure of tex cache hit and tex cache miss. Though the profiler shows such information, in reality both profiler counters are not available on architectures with Compute Capability lesser than 2.0 i.e. even those counters are not measurable on 8400GS on M1060.

Before going ahead and analysing the results of each cache profiler counter, it would be helpful if the throughput of the Matrix Transpose SDK code example of each architecture is displayed:

<table>
<thead>
<tr>
<th>Transpose Kernel name</th>
<th>8400GS Outer/Inner (GB/s)</th>
<th>M1060 Outer/Inner (GB/s)</th>
<th>C2050 Outer/Inner (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple copy</td>
<td>N/A</td>
<td>66.8/68.9</td>
<td>62.5/293.7</td>
</tr>
<tr>
<td>shared memory copy</td>
<td>N/A</td>
<td>44.3/70.8</td>
<td>47.9/124.6</td>
</tr>
<tr>
<td>naive</td>
<td>N/A</td>
<td>1.9/1.9</td>
<td>34.1/62.4</td>
</tr>
<tr>
<td>coalesced</td>
<td>N/A</td>
<td>17.4/17.4</td>
<td>50.5/127.1</td>
</tr>
<tr>
<td>optimised</td>
<td>N/A</td>
<td>17.5/17.5</td>
<td>54.3/189.6</td>
</tr>
<tr>
<td>coarse-grained</td>
<td>N/A</td>
<td>17.6/17.6</td>
<td>54.9/190.9</td>
</tr>
<tr>
<td>fine-grained</td>
<td>N/A</td>
<td>65.6/71.0</td>
<td>56.1/189.6</td>
</tr>
<tr>
<td>diagonal</td>
<td>N/A</td>
<td>28.7/71.4</td>
<td>44.7/189.4</td>
</tr>
</tbody>
</table>

Table 5 Matrix transpose throughput for different kernels on different GPUs

*Outer: Outer loop timing – Inner: Inner loop timing*

The 8400GS does not produce any throughput results because the code addresses only devices with Compute Capability higher or equal to 1.2. Two definite factors that attribute to the vast increase of performance on the Fermi architecture are; the increased number of CUDA cores from 240 on M1060 to 448 on C2050, and the improved cache hierarchy. The profiler produces large sets of data, and specific rows or columns have to be chosen carefully for a specific inspection. Here we are addressing the L1 and L2 cache on the Fermi architecture. Since the diagonal kernel was the most effective transpose optimisation as in *Table 5*, its profiling results were examined. *Figure 8* is the Memory Throughput Analysis tab run on C2050. This tab is not available for devices with Compute Capability lesser than 2.0 i.e. not available for 8400GS or M1060. From *Figure 8*, a certain deduction can be made, is that L1 and L2 caches were relied upon heavily when present, thus playing an integral role in deciding on application performance.
Memory Throughput Analysis for kernel transposeDiagonal on device Tesla C2050 / C2070

- L1 cache read throughput(GB/s): 52.11
- L1 cache global hit ratio (%): 6.55
- Texture cache memory throughput(GB/s): 0.00
- Texture cache hit rate(%): 0.00
- L2 cache texture memory read throughput(GB/s): 0.00
- L2 cache global memory read throughput(GB/s): 47.33
- L2 cache global memory write throughput(GB/s): 25.90
- L2 cache global memory throughput(GB/s): 73.23
- Local memory bus traffic(%): 0.00
- Achieved global memory read throughput(GB/s): 60.35
- Achieved global memory write throughput(GB/s): 33.60
- Achieved global memory throughput(GB/s): 93.95

Figure 8 CUDA Visual Profiler; Memory Throughput Analysis on C2050
3.3 Test 3: Giga-Thread Engine

3.3.1 Concurrent Kernels - Background

A revolutionary capability has been added to the GPU world with the introduction of the Fermi architecture. The ability to maximize the throughput by faster context switching, concurrent kernel execution is now possible on the Fermi architecture. Earlier GPU generations launched GPU kernel calls in a sequential fashion. In other words;

\[
\text{kernel1} \lll \text{kernel2} \lll \text{kernel3}
\]

If there were three CUDA kernel calls inside the main program, any GPU architecture before Fermi, would execute kernel1, followed by kernel2 and finally kernel3. This limitation made sure there was no way to carry out kernel1 and kernel2 in parallel, even if both kernels were totally independent. In terms of performance, that meant that the total execution time would be:

\[
\text{Time}_{\text{Execution}} = \text{kernel1} + \text{kernel2} + \text{kernel3}
\]

The ability to execute CUDA functions simultaneously was there before the Fermi architecture. In M1060 for instance, asynchronous transfer mode is supported. This mode takes place when a CUDA kernel is launched while at the same time, data is being copied to the host from the GPU or vice versa. This upgrade was a way to hide latency and increase performance. This mode is initiated only using CUDA Streams. GPUs with Compute Capability 1.1 or higher can overlap a kernel execution of one stream with a memcopy of another stream, when instructed. However, concurrent kernel launching was only developed on the Fermi architecture.

The Concurrent Kernels CUDA example in the SDK package runs a simple code to test the ability of a GPU device to run launch kernels concurrently. The code uses streams for concurrent execution. GPU devices with compute capability 2.0 (i.e. Fermi) or higher will launch the kernels concurrently; otherwise the kernels will be launched sequentially. The code uses two CUDA kernels and declares eight streams. The first kernel does no real work besides running for a specific number of clock cycles and is run by all running concurrent streams. The second kernel is a parallel reduction kernel that is run by the slowest stream. The last stream to finish the first kernel runs the second kernel and copies data from the GPU to the CPU asynchronously. At the end of the main program, the expected time for the serial execution of the launched kernels, the expected time for the concurrent execution of those kernels and the actual measured time for running those kernels in parallel are printed out.
3.3.2 Concurrent Kernels - Results and Analysis

When using the CUDA Visual Profiler to track the Concurrent Kernels SDK code asynchronous events and parallel kernel launches, a similar behaviour is observed on all three GPU architectures;

![Width Plot](image)

**Figure 9 GPU Time Width Plot; Same behaviour on 8400GS, M1060 and C2050**

Referring to *Figure 9*, it is clear that the launched Streams were run sequentially. On the 8400GS and M1060, the behaviour in *Figure 9* is expected because both of them have Compute Capability lesser than 2.0, which does not support parallel kernel launches. But in *Figure 9*, C2050 is the architecture under test. The only explanation behind the serial behaviour in the figure is that in most cases any profiler adds to the execution a lot of additional events to enable data logging and instrumentation of any program on the device. This explains the stream behaviour seen in *Figure 9* and has the effect of serializing events that would otherwise be asynchronous. The parallel execution of the kernels on C2050 is displayed in the *Table 6*. The only way available at the moment, to measure the performance of concurrent running kernels is by adding timing procedures. This has been added to the parallel kernel launch part of the code and it has been confirmed that only on the Fermi architecture does the kernel streams run in parallel.
<table>
<thead>
<tr>
<th></th>
<th>8400GS</th>
<th>M1060</th>
<th>C2050</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expected time for serial execution of 8</td>
<td>0.080</td>
<td>0.080</td>
<td>0.080</td>
</tr>
<tr>
<td>kernels (s)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expected time for concurrent execution of 8</td>
<td>0.010</td>
<td>0.010</td>
<td>0.010</td>
</tr>
<tr>
<td>kernels (s)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Actual measured time (s)</td>
<td>0.080</td>
<td>0.080</td>
<td>0.010</td>
</tr>
</tbody>
</table>

Table 6 Execution time for Concurrent Kernels
3.4 Test 4: PCI-Express Gen 2.0 Data Transfer

3.4.1 Bandwidth – Background

One of the limiting factors that worry programmers from porting their applications for acceleration on GPUs is mainly concerned with the time spent copying data to and forth the CPU from the GPU, since it is the slowest operation that does no actual work but copy data. The bottleneck here is transferring data between the CPU and the GPU, which happens over the PCI bus. For a CUDA program that must process a huge data set residing on the CPU’s memory, it may take longer to transfer that data to the GPU than to execute the actual computation. The GPU offers the greatest advantage over the CPU for programs which fall in one of two scenarios; the input data is small or there is a large amount of computation relative to the size of the input data.

The new Fermi architecture features the high speed, PCI-Express Gen 2.0 Data Transfer. Unlike PCI Gen 1 used on the G80 systems, PCI Gen 2 is double the bandwidth. According to the PCI website [32], the bit rate has been doubled to satisfy high-bandwidth applications. The website mentions that a PCI Express 1.1 with 8 lanes totals a bandwidth of 4Gbps, which is the same bandwidth attained from a PCI Express Gen 2.0 with 4 lanes. The outcome is substantial savings in platform implementation cost while maintaining the same performance standard.

The Bandwidth SDK code example measures the memcpy function bandwidth of the GPU. Memcopy is a CUDA function that copies data from the CPU’s memory to the GPU’s memory and vice versa. The code measures host to device bandwidth, device to host bandwidth and device to device bandwidth. Those measurements are applied on either pinned or pageable memory. Pinned memory is the CPU memory allocated using the CUDA cudaMallocHost function, which prevents the memory from being swapped out and delivers faster transfer speeds. Non-pinned memory or Pageable memory is memory allocated using the commonly used C malloc function [33]. Pinned memory is much more expensive to allocate and de-allocate but provides higher transfer throughput for larger memory transferals. If pinned memory is allocated in excess amounts, it may result in system performance degradation, since it decreases the amount of memory available to the system for paging.

The code operates on several testing modes. The benchmark tests both pinned and pageable memory transfers, in quick, range and shmoo modes. The quick mode displays basic bandwidth information relating to transfers made (32 Mbytes) from to the host to the device and vice versa. The bandwidth of device to device copies is also included in quick mode. In the range mode, the user specifies at the command line the range of transfer sizes in bytes for bandwidth benchmarking. The last test mode is the shmoo mode, which performs an intense bandwidth test with a large set of transfer values, ranging from 1KB to 64MB.
3.4.2 Bandwidth - Results and Analysis

The shmoo test has been carried out, the results collected, and plotted as shown in *Figure 10* and *Figure 11*. Several important observations were made:

1. The M1060, the predecessor of the Fermi architecture has encountered a close behaviour in terms of bandwidth, when compared to C2050, in both cases pinned and pageable memory modes. The main reason behind the high bandwidth performance of M1060 is because; the M1060 is an embedded module and not a separate GPU card [34]. Though M1060 uses the PCIe Express x16 as in [35, 36], i.e. should be expected to behave closely to how the 8400GS behaves, it is embedded into a server setting that often uses the Onboard Infiniband (QDR) connection for high throughput and low latency as in [37, 38]. Thus behaving in a very close manner to the Fermi architecture.

2. For the pageable memory mode in *Figure 10*, the graphs display data in a more stable fashion and behave as expected. Unlike the pageable memory, the pinned memory graphs in *Figure 4* behave in an unexpected fashion and the results are more unstable. In fact, when the shmoo test is run several times using the pinned memory on all architectures, the bandwidth error is in the range of ± 0.5 GB/s, which is a huge
error. The graphs in Figure 11 represent a mean throughput per run, though the standard deviation is relatively high. Nvidia has warned stating that the pinned memory should not be overused [33]. Unnecessary use of excess amounts of pinned memory can decrease overall system performance because the pinned memory is a limited system resource. The reason behind the instability of the bandwidth results in pinned memory mode cannot be attributed to one specific reason, but the default run mode for the bandwidth application is the pageable memory mode, i.e. the Bandwidth SDK example code was developed with the knowledge of such existing behaviour, which makes the reasons for such behaviour still uncertain.

3. Since M1060 is an embedded module, the fair comparison is the one done between the Fermi architecture C2050 and 8400GS. Taking into account Figure 10 and 11, the increase in bandwidth throughput performance in the Fermi architecture has remarkably improved.
Chapter 4

SHOC

Today, in the HPC arena, heterogeneous computing systems have become one of the novel ways that address the need for continuous performance improvement while managing the pressing challenge of energy efficiency. As these systems become more common, it is of vital importance to be able to compare and contrast architectural designs and programming systems in a fair environment and if possible set a standard for such comparisons. The Scalable Heterogeneous Computing benchmark suite (SHOC) [19] addresses such emerging needs. It is a set of benchmark codes that evaluate the performance of heterogeneous computing devices with non-traditional architectures. This suite is aimed at evaluating the performance of GPUs and multi-core processes, and on testing the industry standard for multi-core programming, OpenCL. The suit supports individual node versions, as well as parallel versions that could be run on larger sized cluster. SHOC includes benchmark implementations in both OpenCL and CUDA in order to provide a comparison of these programming models. The codes used in this dissertation belong to the CUDA C versions. The SHOC benchmark package has three level benchmarks; Level 0: At the lowest level or Level 0, SHOC uses mini-benchmarks to evaluate architectural aspects of a given device, such as measuring bandwidth of transferring data across the PCIe bus to the device and vice versa, measuring bandwidth of memory accesses to various types of GPU device memory including global, shared, and texture memories, and measuring maximum floating point performance of a given device. Level 1: These are higher level benchmarks that evaluate more complex and commonly used applications such as FFTs, MDs, Reductions, Scan and Sorting operations on a given device. Level 2: Includes industrial simulations that evaluate the device in hand for real time performance. Currently, the SHOC package has one such code, which is the S3D turbulent combustion code, implemented in both CUDA and OpenCL.

Most of the SHOC benchmarks included in the package test for both single precision and double precision arithmetic. In all benchmarks, the code runs first the single precision benchmark test, and then checks if the device being used supports double precision arithmetic. If not, the program skips the double precision tests.
4.1 Device Memory

Of all the upgrades that took place with the introduction of the Fermi architecture, improved memory access costs and patterns were the most evident. The Device Memory code, written in CUDA C and OpenCL, available under Level 0 of the SHOC package measures almost all types of memory access costs available on any GPU architecture including global, shared, texture, coalesced memory accesses and many others.

The code has two major parts; the first is measuring texture memory access bandwidth and the second part measures access bandwidth to other memory types.

Texture Memory: Using the texture memory is often a worthy alternative to global memory, especially when it comes to access patterns that prevent appropriate coalescing. The texture memory part of the code measures bandwidth for specific texture access patterns using a two dimensional texture imaged like sized array. The access patterns to the texture memory that are measured are sequential, random and repeated access. For each access pattern, array sizes ranging from 16KB to 4MB are processed. For sequential access, simple repeated linear texel reads are made from texture memory. As for the random access pattern, texels reads are made randomly from texture memory. Finally, the repeated texture reads are cached reads from the texture memory. The repeated reads are only 4KB of texels in size, fitting into texture cache.

Other Memory Types: Other memory access patterns that have their bandwidth measured include global reads, coalesced global reads, local (shared) memory reads, global writes, coalesced global writes and local (shared) memory writes. For all these access patterns, block sizes ranging from 32 to 512 is given to all kernel launches, with block sizes representing the number of threads per block the kernel executes. For the global memory reads kernel, reads are made directly from global memory. Coalesced global memory reads kernel, reads data from global memory are made at strides of 32768 in memory. Shared memory reads kernel declares a shared memory of size 2048, copies global memory data on to it, and then the data is read back from the shared memory into declared registers in turns (iterations) until the whole array in global memory goes through the same operations. Global memory writes, coalesced global memory writes and shared memory writes kernels go through the same operations the read kernels went through, except with the reads switched on to writes.

Tables of how the results were tabulated are shown below. Several rows and columns were ignored in order to enhance readability. The most remarkable bandwidth improvements are noticed in three areas. Texture cache hits and read/writes to local memory. The explanation behind these improvements is the newly added L1/L2 cache hierarchy that has been added to the Fermi architecture. Being a configurable 64KB cache structure, L1 was used alongside with the shared memory, and the texture load granularity is now 128 bytes at L1 level, and 32 bytes at L2 and global memory level, boosting performance extensively.
8400GS does not have texture memory. M1060 has a 16KB shared memory in each SM. This does not function as a true cache, but has to be managed via the software application by the developer. On the Fermi architecture, C2050, the size of the shared memory has increased to 64KB and it can now operate as a real L1 cache. On C2050, all four Graphical Processor Clusters (GPCs) share a large 768KB L2 cache. Each SM has four texture units, each capable of 1 texture address and 4 texture sample operations. There is now more texture sampling units but fewer texture addressing units, explaining the boost in performance of the texture memory bandwidth in Figure 6. C2050’s L1 cache line size is 128 bytes. If threads inside a warp access different cache lines, even though if they reside all in L1’s cache, there is a 1/32 L1 theoretical bandwidth, because each thread issues one 128 byte transaction to get 4 byte float value. Texture cache doesn’t have such a problem. Their cache line size is 32 bytes, so 4 byte float values can be accessed independently. A float penalty is 1/4 which is a lot better than L1 cache.
4.2 MaxFlops

Double precision arithmetic is a core to many major HPC applications. If there is one capability added newly to the Fermi architecture that Nvidia won’t stop boasting around about, it is the double precision performance. Nvidia states on their website that Fermi can run double precision up to eight times faster when compared to the GT200 series GPUs, Fermi’s predecessors [39]. Not only that, but Fermi GPUs have managed to run double precision operations at half the speed the single precision operations run. In the previous GPU generations, double precision operations were running at one-tenth the speed of a single precision operation, and the reason that attributed to such performance was because, double precision was handled by one dedicated unit per SM, while at the same time, it had eight single precision units. But now the Fermi architecture is capable of running 16 double precision operations per SM per clock cycle.

The MaxFlops code, under Level 0 in the SHOC package, is a code written specifically to measure the maximum floating point operations per second a GPU is capable of handling. The code has six main kernel operations; addition, multiply-adds, hand tuned multiply-adds, multiplication, multiplication and multiply-adds, and hand tuned multiplication & multiply-adds. Every kernel version of each of those operations calculates the floating point throughput using both single precision and double precision. Also every version of those operations, operate on 1, 2, 4 and 8 independent stream versions of the same function in turns. See Figure 13 below;

![Code snippets for Addition operation kernels](image)

Figure 13 Code snippets for Addition operation kernels

In Figure 14, the MaxFlops code was run on C2050 and M1060. DP stands for double precision and SP stands for single precision. The timing was based on timing independent CUDA launched kernels, but the timings in the figure represent the fastest Kernel running times. See Appendix B for exact numbers.

![Maximum DP and SP performance](image)

Figure 14 Maximum DP and SP performance
4.3 S3D

In a rather short period of time, GPUs have evolved from an unknown, highly specialized hardware component into an extraordinarily flexible and powerful parallel processor. With the introduction of Fermi, a similar step has been taken in the forward direction in the world of GPUs. The GF100 series of the Tesla GPUs support first class features for technical and enterprise computing. Testing industrialized applications on the Fermi architecture is an integral step required to prove that Fermi is not one dimensional. It is not only built for HPC scientific purposes, but might as well fit perfectly with real commercial, financial and industrial level applications. In the SHOC benchmark suite, one such fully developed application exists, and could be used to evaluate Fermi’s new capabilities.

The S3D turbulent combustion simulation application is an immensely parallel direct numerical solver for the full compressible motion of fluid substances, dynamics, total energy, species and mass continuity equations coupled with in detailed chemistry [40, 41]. The S3D has been ported to GPUs by [42] and is developed in both CUDA C and OpenCL. The S3D code is available under Level 2 of the SHOC package. A brief summary of the overall structure of the SD3 code should have been given in the code, in order to have a better understanding of the results. The CUDA kernel functions were poorly documented, in that not enough explanation was provided on what exactly each kernel does or what was its purpose. That is why a lot of time has been spent analysing the code, to get a thorough image of how the program is supposed to function. Two core parts were involved in the analysis process; the main function, which initialised the variables, launched the kernels and timed the whole process and, the other part were the kernel functions themselves. The kernel functions will not be explored further, as the computations are very complex and program specific.

Main function:

Like every other CUDA main function, this function starts by declaring host and device variables. The host variables are initialised and memory is allocated for the device memory. A point worthy of mentioning is that all the memory that allocated for the GPU is pinned or page-locked. Refer to section 3.4.1 for further details. The code then declares two streams, i.e. preparing the application for either asynchronous transfer, kernel parallel execution or both. cudaMemcpyAsync() is then used to copy data from the host to the function. Three important facts about this function should be taken into account. First is that this is an asynchronous CUDA function in relation to the host. In other words, the call may actually return before the copy transaction is complete. Second, only pinned memory is an accepted input parameter to the function, and it returns an error if the memory is pageable. Finally, the cudaMemcpyAsync() function actually takes a stream argument, which means that if two different streams are passed to two cudaMemcpyAsync() functions, these operations happen in parallel. Once the data is inside the device memory, the kernel functions are called inside a loop. This
loop has ten iterations, signifying the number of trials this test runs for. Inside the loop, two streams are used during different kernel calls. If two kernels are launched having the same stream call, they occur sequentially, otherwise in parallel. After the loop exits memory is de-allocated on both host and device. Certainly, every step in the program is timed for further analysis.

Running the S3D code using the Compute Visual Profiler did not complete successfully. The code is profiled completely on C2050, giving _qssa_kernel-0_ to be the busiest GPU kernel launch with occupying 3.56% of the time. On M1060, the profiling is interrupted by an error, which informs of rows being dropped out i.e. profiling is missing data on M1060. See Figure 15:

![Profiling Error](image)

Figure 15 Profilers Dropped Error

Searching to a similar behaviour that has been faced by the CUDA developer’s community did not result in any explanation to such behaviour. From the incomplete data produced by the profiler _ratt_kernel-0_ consumes the most GPU running time of 17.5%. Since the profiler session did not complete, the M1060 results were deemed meaningless. As mentioned earlier, the 8400GS does not support DP, both S3D-DP and S3D-DP_PCIe (transfer time + execution time) produced N/A results. The graphs do not represent such variability in the code results between C2050 and M1060. Referring to section 4.2, it is clear from Figure 16 that DP operations run for almost half time as the SP operations on both architectures. In other words, SP operations run twice as fast, which is expected, as single precision floating point arithmetic is processed faster generally on all GPU architectures. Also it is clear the PCIe transfer times are also not an important factor, as it does not alter the performance to a great extent on both architectures. Analysing the code, it was verified that shared memory was not used as the L1 global cache hit ratio was 0%. The slight performance boost between C2050 and M1060 could be accounted for the extensive usage of L2 cache, as L2 global memory throughput reached up to 92.83 GB/s for the _qssa_kernel-0_ kernel on C2050. See Figure 17.

![Figure 16](image)

Figure 16 DP and SP Execution Times

![Figure 17](image)

Figure 17 L1 and L2 Cache Throughput
The graph in Figure 16 is from a set of tabulated timing results from running the S3D code. The code timed bandwidth operations for DP, SP, DP including PCIe bandwidth and SP including PCIe bandwidth.

![Figure 16 S3D Throughput](image)

**Figure 16 S3D Throughput**

**Memory Throughput Analysis for kernel qssa_kernel-0 on device Tesla C2050 / C2070**

- L1 cache read throughput (GB/s): 64.84
- L1 cache global hit ratio (%): 0.00
- Texture cache memory throughput (GB/s): 0.00
- Texture cache hit rate (%): 0.00
- L2 cache texture memory read throughput (GB/s): 0.00
- L2 cache global memory read throughput (GB/s): 64.23
- L2 cache global memory write throughput (GB/s): 28.59
- L2 cache global memory throughput (GB/s): 92.83
- Local memory bus traffic (%): 53.05

**Figure 17 S3D Memory Throughput Analysis**
Chapter 5

NPB

The NAS Parallel Benchmark (NPB) suit is set of applications designed to assist in evaluating the performance of parallel supercomputers [21]. The suit is developed and maintained by the NASA Advanced Supercomputing (NAS) Division located at the NASA Ames Research Centre, in California. It provides computing assets for various major NASA developments. The NPB have become a standard in distributed scientific computing and many middleware applications are generally based on them.

There are three major NPB releases; (NPB 2.3 was used in this dissertation)

NPB 1

NPB 1 consisted of eight benchmarks, with each benchmark defined in two problem sizes, Class $A$ and Class $B$. The codes were written in Fortran 77. The benchmarks also used a problem size Class $S$ but were not intended for benchmarking purposes. NPB 1 has two major flaws. One, its "paper-and-pencil" specification led computer vendors to highly tune their implementations with impractical performances that made it not possible for scientific programmers to attain. The other flaw were that many of these implementations were exclusive to the vendor and not publicly accessible, hiding the optimisation techniques that were used on those implementations.

NPB 2

NPB2 has the source code implementations for five out of eight benchmarks available in NPB 1. NPB 2 complements and does not replace NPB 1, by adding to the benchmarks a new problem size, Class $C$. Most importantly the rules for submitting the benchmark results were redefined. The new rules stated explicitly that all output files, modified source files and build scripts were publicly available for modifications and reproducibility of those results. NPB 2.2 had two more benchmarks added to the initial NPB 2 version [21]. NPB 2.3 is the first complete implementation of the NPB benchmarks written in MPI, and had a new class size added to it, Class $W$ for systems low on memory. Class $D$, a larger problem size was added in NPB 2.4, which had a modified version of the MPI implementation.
NPB 3

NPB 3 used the same MPI implementations from NPB 2, and came up with newer framework implementations such as OpenMP, Java and High Performance FORTRAN (HPF) [21]. Those newer versions were based on the serial codes from NPB 2.3, but with extra added optimisations. The NPB 3.3 introduced a Class E problem size. Also a set of “multi-zone” benchmarks that include serial and parallel versions have been added to the package. Those benchmarks are meant to test multi-level and hybrid parallelization paradigms. MPI is used for the coarse-grain parallelism, combined with OpenMP for the loop-level parallelism.

Six classes of problem sizes were mentioned; $S$, $W$, $A$, $B$, $C$ and $D$. Class $S$ is the easiest problem size and is for testing purposes only, while Class $D$ is the hardest. These classes are available for all NPB benchmarks.

The NPB benchmarks are derived from computational fluid dynamics applications and contain five kernels and three generic applications.

The five kernels are [21];

EP: An embarrassingly parallel code that provides an estimate of the highest achievable limits for floating point performance without significant processor communication. The kernel generates pseudo-random floating point Gaussian variables using the Marsaglia polar method.

MG: A basic multi-grid kernel that involves highly structured long distance communications and examines short and long space data communication. The kernel estimates a solution to the discrete Poisson problem.

CG: The Conjugate Gradient kernel. This was the code chosen for porting to GPUs. Refer to section 5.1.

FT: Figures a 3D partial differential equation (PDE) solution using the Fast Fourier Transform (FFT). It includes thorough tests of communication performance.

IS: IS is a large integer sorting kernel based on the bucket sorting algorithm. It tests integer computation speed and also communication performance.
The Conjugate Gradient (CG) benchmark approximates the minimum eigenvalue of a large sparse symmetric positive-definite matrix using the inverse iteration coupled with the conjugate gradient function as a subroutine for solving systems of linear equations. The CG code is a perfect example of unstructured grid computations. It tests unequal long distance communications and employs unstructured matrix vector multiplications. The CG code is totally dependent on communications, in that the parallel versions of it perform initiate many synchronization barriers between the processors. Figure 18 demonstrates such behaviour;

![Diagram of CG parallel synchronisation behaviour](image)

Figure 18 The CG parallel synchronisation behaviour

One of the main advantages of using GPUs is its high number of processors, so it definitely follows that the launched kernels have to exhibit a high degree of parallelism in order to be successful on a GPU platform. This makes it very difficult and performance degrading for GPUs to handle unstructured kernels, or those with complex patterns of data dependencies, such as the case with the CG implementation. Certainly, in conditions with irregular control flow, threads can run sequentially, disturbing application performance to a great extent. On GPUs, memory access times can differ by several orders of magnitude based on access pattern and type of memory. In other words for example, an access to shared block memory is 100’s of cycles faster than an access to global memory. As such, kernels must often be chosen based on memory access patterns that are more uniform in nature. But as stated by Nvidia and otherwise, memory access patterns and bandwidth have seen dramatic upgrades on the Fermi architecture, which makes CG a good candidate for porting on to CUDA. Not because CG will offer any performance gains on the GPU architecture, but would allow testing and evaluating memory access patterns, bandwidth, atomic instrinsics and synchronisation barrier performance on the Fermi architecture. As shown in Figure 18, the communication pattern is not at all uniform, and the code has a lot of data dependencies.
5.2 Initial Code and Profiling

The CG code that was chosen for porting on to GPUs for acceleration is developed by Real World Computing (RWCP) and is an OpenMP C version based upon the serial FORTRAN versions in NPB 2.3 serial implementations [43]. Two core functions in the CG code were `makea` and `conj_grad`. The `makea` function creates a sparse matrix with a predefined sparsity distribution. The `conj_grad` is the conjugate gradient method that has an iteration loop, which consists of one sparse-matrix vector multiplication, four reduction sums and several paxpy calls. The code was profiled using gprof 2.17.50 [44].

```
Flat profile:

Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th></th>
<th>% cumulative</th>
<th>self</th>
<th>self</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>time seconds</td>
<td>seconds</td>
<td>calls</td>
<td>s/call</td>
<td>s/call name</td>
</tr>
<tr>
<td>99.51</td>
<td>540.44</td>
<td>76</td>
<td>71.11</td>
<td>conj_grad</td>
</tr>
<tr>
<td>0.40</td>
<td>542.62</td>
<td>1</td>
<td>21.8</td>
<td>sparse</td>
</tr>
<tr>
<td>0.10</td>
<td>543.17</td>
<td>0.55</td>
<td>5.5</td>
<td>makea</td>
</tr>
<tr>
<td>0.03</td>
<td>543.35</td>
<td>0.18</td>
<td></td>
<td>main</td>
</tr>
<tr>
<td>0.03</td>
<td>543.52</td>
<td>0.17</td>
<td></td>
<td>rand1c</td>
</tr>
<tr>
<td>0.01</td>
<td>543.58</td>
<td>0.06</td>
<td>150000</td>
<td>sprmvx</td>
</tr>
<tr>
<td>0.00</td>
<td>543.59</td>
<td>0.01</td>
<td>3936140</td>
<td>icmvrt</td>
</tr>
<tr>
<td>0.00</td>
<td>543.60</td>
<td>0.01</td>
<td>150000</td>
<td>vecset</td>
</tr>
<tr>
<td>0.00</td>
<td>543.61</td>
<td>0.01</td>
<td></td>
<td>c_print_results</td>
</tr>
</tbody>
</table>
```

Table 7 Partial profile of CG

The CG code has been compiled with the Class C problem size, and the result of the profiled data is observed in Table 7. It shows that the `conj_grad` method almost takes 99% of the application running time, making it the most suited function for acceleration on the GPU. In addition, the `conj_grad` method has four reduction kernels, which is what is going to be used for evaluating GPU atomic calls or synchronisation barriers, when the code is ported.
5.3 Porting CG

The simplest and most naïve way to convert a C function into CUDA code, is by just adding the `__global__` intrinsic CUDA operator before the name of the C function. This should be the first step taken before any optimisations could be applied to the newly converted CUDA function. Once the code output is verified, changes to the structure of the CUDA code can then be made. However, as mentioned in section 5.1, that the CG code, mainly the `conj_grad` method, is unstructured and has non uniform access patterns. So just adding the `__global__` operator would definitely produce wrong results. The main problem lies in the reduction operators added to the `conj_grad` method, from the OpenMP version, which was based on a serial version that just looped through the whole array to do the summing procedures. It was critical to understand at this point, there was no other way but using synchronisation barriers to compute the sum. Once the sum was computed, it was used as an input for further operations across all processors. But until today, CUDA does not support global synchronisation barriers inside a CUDA kernel i.e. a function in CUDA that forces all threads in all blocks to halt execution at a certain point, does not exist. The reason behind this is that it is expensive from a hardware perspective to do this for GPUs with high processor count. It would also force developers to run fewer blocks, to avoid a deadlock, which may reduce overall efficiency. See Figure 19 and the code snippet on the following page.

![Figure 19 Synchronisation never achieved, Deadlock](image)

Each GPU SM holds a maximum number of X thread blocks at any given point of time, depending on the architecture. So at most, the number of SMs available multiplied by X number of blocks could be active at any given point of time. This means that the only way to cope up with such limitation is reducing the number of blocks, in order to avoid
the deadlock situation, which would reduce GPU occupancy percentage, leading to a poor execution performance. Thus, synchronising variables inside a kernel, to further use global computed values on all threads available, is out of the picture and no more a choice the porting could use. Other options could be using optimised reduction CUDA kernels, atomic functions or serialised kernel launches.

```
static void conj_grad (....)
{
    ....
    #pragma omp for reduction(+:rho)
    for (j = 1; j = lastcol-firstcol+1; j++) {
        rho = rho + x[j]*x[j];
    }
    ....
    for (cgit = 1; cgit = cgitmax; cgit++) {
        ....
        #pragma omp for reduction(+:d)
        for (j = 1; j = lastcol-firstcol+1; j++) {
            d = d + p[j]*q[j];
        }
        #pragma omp for reduction(+:rho)
        for (j = 1; j = lastcol-firstcol+1; j++) {
            rho = rho + r[j]*r[j];
        }
        #pragma omp single
        beta = rho / rho0;
        #pragma omp for
        for (j = 1; j = lastcol-firstcol+1; j++) {
            p[j] = r[j] + beta*p[j];
        }
    }  /* end of do cgit=1,cgitmax */
    #pragma omp for reduction(+:sum) private(d)
    for (j = 1; j = lastcol-firstcol+1; j++) {
        ....
    }
    ....
}
```

Going through extensive research about what has been experimented on in regards to available GPU synchronisation options; one of the most recent papers published in this track is [45] where Xai and Feng propose three GPU synchronisation mechanisms. The simple synchronisation, tree based synchronisation and the lock-free synchronisation. Both the simple synchronisation and the tree based synchronisation are lock-based designs and make use of atomic operations and mutex variables. The lock-free synchronisation avoids the use of atomic operations, and the authors prove in the paper that it’s the most effective method GPU synchronisation method, from the performance perspective. Other options considered were replacing the reduction kernels with optimised reduction kernels used in the Nvidia’s SDK package. A serious flaw existed with all these options. All of them presumed that after the synchronisation operation, the reduced value would be saved on global memory and then the function would return. In the `conj_grad` method, this is not the case. After the reduction methods are called, the reduced value is further used by all processors available, in the OpenMP version i.e. if this were to be implemented on the GPU, the synchronisation should take place, the reduced value saved on global memory, and that value is advanced to all the threads in all blocks available. This solution is not plausible, not even on the most recent Fermi architecture with the most current Compute Capability. The only reliable and supported method available currently for a kernel-wide synchronization is another kernel launch.
Concurrent kernels could have been thought of as an idea, since it appears to solve an important issue. The time spent in launching one kernel after the other is parallelised and the performance is not so much affected. But this will produce wrong results, since every kernel launched is dependent on the results from the previous kernel launch. As such, finally it was decided that the `conj_grad` method will be cut into eight separate CUDA kernels, where the separation is based on the synchronisation point. CUDA kernel launches are asynchronous yes, but in relation to the host. A CUDA kernel will run on the device once all previous CUDA calls have finished. That implicit barrier is used as a synchronisation point between every kernel launch. Certainly, the time spent in launching every kernel after the other affects the performance to a huge extent. But an important reminder; the aim here is not performance, but timing how fast did context switching improve on the Fermi architecture.

The `conj_grad` code has been ported into eight different CUDA kernels. The kernels were called from within the `conj_grad` iteration loop. The code has been timed using CUDA event handlers. The overall execution time of the CUDA code was timed, as well as kernel launch overheads. To calculate the kernel launch overheads, an empty bodied kernel was launched with one CUDA thread inside a loop of N iterations. The reason why kernel launch overheads were calculated was to get an approximate value of how much that figure improved on the Fermi architecture compared with its GPU predecessors in regards with kernel launch overheads. See Figure 20.

```c
for (i = 0; i < N; i++) {
    cudaEventRecord(start, 0);
    empty_kernel<<<1,1>>>(0);
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&time, start, stop);
    time += x_time;
}
```

Figure 20 Calculating Kernel launch overhead
5.4 Results and Analysis

The overall execution time of the ported CUDA `conj_grad` method was timed (eight kernels) on C2050 and M1060. With Compute capability 1.1, 8400GS did not respond any of the tests, as all doubles were demoted to floats and precision was lost leading to invalid results. See Table 8. The GPU code performance has been compared with a sequential CPU run on the Intel(R) Xeon® CPU E5504, available on the NESS system.

<table>
<thead>
<tr>
<th>CLASS S (s)</th>
<th>Intel(R) Xeon® CPU E5504</th>
<th>8400GS</th>
<th>M1060</th>
<th>C2050</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.07</td>
<td>N/A</td>
<td>1.01</td>
<td>0.48</td>
</tr>
<tr>
<td>CLASS A (s)</td>
<td>2.17</td>
<td>N/A</td>
<td>3.61</td>
<td>1.82</td>
</tr>
<tr>
<td>CLASS C (s)</td>
<td>353.11</td>
<td>N/A</td>
<td>191.23</td>
<td>152.57</td>
</tr>
</tbody>
</table>

Table 8 Overall Execution times of CG kernels

The code performs well, without any optimisation techniques added to it. The aim was to evaluate how the architecture would cope up with code that was just simply ported into CUDA code. This meant that all data was written to and from global memory directly. Using the code in such a way allowed to evaluate the achieved overall global and cache bandwidth improvements on Fermi, supporting the results found in previous sections. Using the Compute Visual profiler to profile the test runs for Class C problem sizes, `conj_gad_3` kernel was the most GPU time consuming. As shown in Figure 21, the overall achieved global memory bandwidth for the `conj_gad_3` function on M1060 is 38.7 GB/s, while on C2050, it has a bandwidth of 133 GB/s, thus increasing the performance of the code to a huge extent.

![Figure 21 Global memory and L1 cache Throughput](image)

38
Kernel launching was the solution chosen to overcome explicit synchronisation barrier issues. As such, the kernel launch overheads were timed on all the three GPU architectures used. See Figure 20. Though the code was not run on the 8400GS, it was beneficial to time the kernel launches on also this architecture, only to view how much different is it for the Fermi architecture. When the code was timed with the code in Figure 20, the timing results were very close on all the architectures. They all showed a kernel launch overhead of 6.5us. On the C2050 this is an acceptable result, as it was mentioned on the Nvidia website [17] that the throughput was improved by faster context switching and reduced kernel launch overheads. But both M1060 and 8400GS could not logically have such small and similar kernel launch overheads.

The code in Figure 20 was modified to have a larger loop (1000000 iterations), and the kernel launch had 512 threads and 512 blocks instead of just one. Since an empty kernel has been with only 1 thread, only a part of the kernel launch overhead was measured, which increases with the number of threads per blocks. The new kernel launch was;

```c
empty_kernel<<<512,512>>>();
```

The results obtained from such a modification were the following; 8400GS needed about 84.7us per kernel launch, M1060 about 7.9us, while C2050 needed 5.2us seconds per kernel launch. The reason why M1060 and C2050 have close difference is that because M1060 is an embedded module, see section 3.4.2, the PCI latencies of C2050 hide how much Fermi have improved in regards to kernel launch overheads. Factors that affect or hide such low calculated overheads include, host operating system and hardware characteristics that could easily mask performance differences between the cards, operating system signal/interrupt, driver and PCI express bus latencies. Reducing the latency performance of something that is not close to the latency bottleneck of the total system won't have much effect on the overall latency that is measured. Also the kernel body, the number of parameters inside a kernel call, are all elements that play a role in a kernel launch overhead. In short, since the kernel launch overhead on the C2050 is relatively small compared to M1060 and 8400GS, having serialised kernel launches to solve the synchronisation issues insures that high performance can still be achieved on data-dependant parallel applications. At the large sized Class C problem, the CUDA conj_grad code on C2050 performed ~2.5X times faster than an Intel(R) Xeon® CPU E5504 processor, without any added optimisation techniques
Chapter 6

Conclusion

This dissertation analysed some of the most prominent features of the Fermi architecture. Using the SDK benchmark suite, three codes were used. The Scan code, which tested the increased CUDA cores capability on Fermi. It verified that compute intensive operations operate at a much higher bandwidth on C2050, with a ~7X and ~35X of speedup, when compared to the M1060 and 8400GS respectively. The Transpose code example was used to analyse the memory hierarchy on C2050, M1060 and 8400GS. Because 8400GS has a compute capability of 1.1, it did not support double precision and did not pass many codes’ validation tests necessary to produce results. The new L2 cache was used heavily in the Transpose code test run on C2050, with a global memory throughput of 73%, boosting up the code’s performance. Concurrent Kernels code tested on all three GPU architectures for the ability to run kernels in parallel, and C2050 was the only architecture that ran the eight streams in parallel with an execution time of 0.10s, compared to a serialised run on M1060 and 8400GS that took 0.80s. The last sdk code that was used was the Bandwidth code. It measured the PCIe cables bandwidth for data transfer to and from the device to the host. Results showed that C2050 and M1060 operated on close bandwidth performance, as M1060 is an embedded module, and does not rely on PCI cables for data transfer. The cumulative bandwidth performance increase on C2050 compared to 8400GS was in the orders of magnitude larger.

The codes used from the SHOC benchmark suite were more thorough. The Device Memory code tested the memory access patterns for different types of memories on the tested GPU architectures. C2050 performed exceptionally well with texture memory cache hit ratio, having a 17.5X speedup on a 4MB array size compared to the M1060. 8400GS did not include texture memory. The MaxFlops code in the SHOC package was used to test Fermi’s exceptional DP performance boosts. Results were compared to many sources including Nvidia, and it was confirmed that DP floating point arithmetic operations take now as half time as needed for a single floating point arithmetic operation, and is ~8X times faster than M1060. 8400GS does not support DP operations. The last SHOC code used was the S3D turbulent combustion and mainly tested industrial level code performance. On C2050, the code had a speedup of ~2X compared to M1060 and ~30X compared to 8400Gs’s single precision run.
The last phase of the document discussed the ported implementation of the Conjugate Gradient code for the NPB benchmark suite and analysed the results. The code mainly addressed synchronisation aspects on the Fermi architecture and GPUs in general. It was confirmed that global synchronisation across all blocks in a grid launched is not supported on GPUs yet. The code times the overall execution time and the serialised kernel launch overheads, since that was the only solution to get around synchronisation issues. Kernel launch overheads were ~16X slower on 8400GS compared to C2050, while M1060 had closer results to C2050.

6.1 Future Work

The amount of data generated trying to analyse Fermi were huge, and took most of the time that was intended for analysing Fermi more thoroughly. Future work could include comparing Fermi’s results to multi-core and many-core parallel versions of MPI and OpenMP, and evaluating performance levels. Other evaluations may involve parallel development frameworks such as CUDA Fortran, OpenCL and PGI directives, to be further analysed and evaluated on the Fermi architecture.
References


[20] SHOC. *SHOC License.*


[21] NASA. *NAS Parallel Benchmarks.*


[23] Li-Wen Chang et al. *Parallel Implementation Of Multi-Dimensional Ensemble Empirical Mode Decomposition.*


[27] Mark Harris. *Parallel Prefix Sum (Scan) with CUDA.*


[29] Greg Ruetsch et al. *Optimizing Matrix Transpose in CUDA.*


[31] Nvidia. *CUDA Toolkit 4.0.*


[34] Nvidia. *Tesla M1060 Processor.*


[37] PHYS. *The Bull bullx systems.*


45
[43] PUDN. *NAS Parallel Benchmarks 2.3 OpenMP C versions - CG.*

[44] School of Computing, The University of UTAH. *GNU gprof.*

[45] Shucai Xia et al. *Inter-Block GPU Communication via Fast Barrier Synchronization.*