GPU Acceleration of a
Theoretical Particle Physics Application

Karthee Sivalingam

August 27, 2010

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2010
Abstract

Graphics processing units (GPU) are commodity processing units in video cards used for generating graphics in High resolution. They provide greater computational power than most commodity CPUs. Particle Physics experiments require simulations to understand and analyse the results. GPUs can be used to accelerate many such applications. In this thesis, acceleration and optimization of Theoretical Particle Physics applications using GPU is studied.

Many programming models support programming of GPUs. CUDAC and directive based approach (PGI) are evaluated for performance and programmability. Currently, CUDAC is the best approach for programming GPUs, as the compilers for directive based approach are not fully developed and supported. But with standardization and improvements, the directive based approach appears to be the best and productive way forward for accelerating application using GPUs.

The recent hardware enhancements in Tesla series (Tesla C2050) from NVIDIA have improved double precision support. Also memory management with hardware caching makes it easy to optimize for performance and reduce memory latency. For the Particle Physics application investigated in this thesis, a speedup of 25 (when compared to the host-Intel(R) Xeon(R) CPU E5504 @ 2.00GHz) was achieved on Tesla 2050 using CUDAC. The PGI Accelerator compiler also shows similar performance when accelerated using directives.
Contents

Chapter 1 Introduction .................................................................................................................... 1
Chapter 2 Background and Literature Review ............................................................................... 2
  2.1 Particle Physics ..................................................................................................................... 2
    2.1.1 Theoretical Particle Physics and Standard Model ...................................................... 2
    2.1.2 Lattice QCD Theory ..................................................................................................... 3
    2.1.3 Lattice Perturbation Theory and Vertex function Calculation ..................................... 3
  2.2 GPGPU .................................................................................................................................. 4
    2.2.1 GPU Architecture .......................................................................................................... 5
    2.2.2 CUDA Programming Model ......................................................................................... 8
    2.2.3 CUDAC .......................................................................................................................... 9
    2.2.4 PGI FORTRAN and C accelerator .............................................................................. 10
    2.2.5 PGI Accelerator Programming Model ........................................................................ 10
    2.2.6 OpenCL ........................................................................................................................ 12
    2.2.7 CUDA FORTRAN ...................................................................................................... 12
  2.3 Others work ......................................................................................................................... 12
Chapter 3 Analysis and Design ..................................................................................................... 14
  3.1 Current Design .................................................................................................................... 14
  3.2 Optimization ........................................................................................................................ 16
    3.2.1 Memory Optimization ................................................................................................. 16
  3.3 Thread Scheduling .............................................................................................................. 20
  3.4 Parallel Decomposition ....................................................................................................... 22
  3.5 Instruction Optimization ..................................................................................................... 22
List of Tables

Table 1 Comparing Tesla M1060 and Tesla C2050 Hardware specifications NA-Not available; * - configurable; ............................................................................................................. 7

Table 2 Table showing the system hardware of different systems used ........................................... 24

Table 3 Profile counters and description ............................................................................................ 59

Table 4 Hardware specification of GPUs in Ness, Fermi and Daresbury systems ................... 61

Table 5 GPU kernel execution time in seconds for CUDAC code version on Ness; increasing npoints; nterms=8000 (includes time to allocate and copy memory, excludes time to initialize device) ........................................................................................................................................... 73

Table 6 GPU kernel execution time in seconds for PGI code version on Fermi; increasing npoints; nterms=8000 (includes time to allocate and copy memory, excludes time to initialize device) ........................................................................................................................................... 73

Table 7 GPU kernel execution time in seconds for code versions on Fermi; increasing npoints; nterms=8000 (includes time to allocate and copy memory, excludes time to initialize device). Initial, 2-D and Block D refers to the decompositions used ................................................................................................................ 74

Table 8 GPU kernel execution time in seconds for code version on Ness; increasing npoints; nterms=8000 (includes time to allocate and copy memory, excludes time to initialize device). Initial, 2-D and Block D refers to the decompositions used ................................................................................................................ 74

Table 9 GPU kernel execution time in seconds for code version with spin and color changes on Fermi; increasing npoints; nterms=8000 (includes time to allocate and copy memory, excludes time to initialize device). ................................................................................................................ 75

Table 10 GPU kernel execution time in seconds for code version with spin and color changes on Fermi; increasing npoints; nterms=8000 (includes time to allocate and copy memory, excludes time to initialize device). ................................................................................................................ 75

Table 11 GPU kernel execution time in seconds for code version on Fermi; increasing nterms; npoints=4096; (includes time to allocate and copy memory, excludes time to initialize device) ................................................................................................................ 75

Table 12 GPU kernel execution time in seconds for code version on Ness; increasing nterms; npoints=4096; (includes time to allocate and copy memory, excludes time to initialize device) ................................................................................................................ 76
List of Figures

Figure 1 A graphical representation of vertex ................................................................. 4
Figure 2 GPU Architecture showing Symmetric Multiprocessors (SM) and Scalar Processors (SP) ................................................................. 6
Figure 3 Cuda programming model and Memory Hierarchy .................................................. 9
Figure 4 Code Translation in PGI Programming ................................................................. 11
Figure 5 Speedup on Ness (compared to host) for different configurations of number of points and nterms .................................................................................. 15
Figure 6 Profile counter plot of the initial code .................................................................... 25
Figure 7 Profile Counter Plot of code version with direct global memory accesses .......... 26
Figure 8 Parallel Decomposition used in the initial code ..................................................... 27
Figure 9 2-D Decomposition uses 4 threads to compute for a point ..................................... 28
Figure 10 Profile Counter Plot of code that uses 2-D decomposition .................................. 29
Figure 11 Profile Counter Plot of code that uses all Shared memory in Complex format ....... 31
Figure 12 Profile Counter Plot of code that uses two double to represent a complex .......... 32
Figure 13 Profile Counter Plot showing better coalescing when thread index are interchanged. ........................................................................................................... 33
Figure 14 Block decomposition showing block of 4 threads working on a single point ........ 34
Figure 15 Profile Counter Plot of code that uses one block of thread for a point ............... 35
Figure 16 Speedup on Ness (compared to host) for different CUDAC Code versions ............ 36
Figure 17 Speedup on Fermi (compared to host) for CUDAC code that using Initial/2-D/Block decomposition ............................................................................................................. 39
Figure 18 Speedup on Fermi (compared to host) for application with Spin/Col changes ....... 41
Figure 19 Speedup on Ness (compared to host) for application with Spin/Col changes ........ 42
Figure 20 Speedup on Fermi (compared to the host) for different PGI accelerated code versions ................................................................. 48

Figure 21 Speedup (compared to Ness CPU) on Fermi and Ness for increasing number of points ............................................................................................................................................................................................................. 51

Figure 22 Speedup on Fermi (compared to host) for increasing number of monomials. ............ 53

Figure 23 Speedup on Ness (compared to host) for increasing number of monomials. .......... 54
Acknowledgements

I am very grateful to Dr Alan Gray (EPCC) and Dr Alistair Hart (Cray) for their support, advice and supervision during this project. They showed confidence in me from the start and their motivation greatly helped me. I thank all the teaching and non teaching staff associated with the MSc, High Performance Computing course at University of Edinburgh. It has been a delight. I thank the EPCC and their support staff for providing good facilities and support. I thank Dave Cable and STFC Daresbury laboratory for providing access and support to their GPU systems. I am thankful to my fellow students at MSc High Performance Computing for helping me all along.
Chapter 1

Introduction

GPUs (Graphic Processing Units) are processing units that are optimized for graphics processing. With the growth of the gaming industry, the performance of these devices has improved. These devices can also be used for scientific computing. GPU shows better speed and performance than most CPUs for floating point arithmetic. This trend is improving as the CPUs have hit the power wall. In recent years there has been a large interest in using GPUs for accelerating HPC applications.

Applications can be accelerated by porting few kernels to GPU. The porting of few kernels with high computation to GPU is not a straight-forward simple procedure and not all applications are suited for GPU acceleration. GPU and CPU don't share a common memory. This means that the input and output data has to be transferred across the memories through the network bus. The SIMD architecture with hundreds of cores, hardware thread management and software memory management makes these devices very powerful and also hard to program. Currently these units can be programmed using many languages like CUDAC, OpenCL and PGI Accelerator directives.

Physics has helped us understand the basic particles of matter that forms the universe. The research of fundamental particles has been in progress for centuries. Many experiments like Large Hadron Collider at CERN are being carried out to better understand the same. These experiments need to be complemented with simulations for further study. These simulations require huge computing power and the need for high performance, low cost computing systems with lesser power requirement has increased. Many such particle physics simulations and applications show many characteristics that make them suitable for GPU acceleration.

The aim of this thesis is to optimize a particle physics application that has already been ported for GPU acceleration by Dr Alistair Hart. Various techniques for optimization will be studied and analyzed. This application is currently programmed using CUDAC. Other programming models like the PGI Accelerator directives will be used and evaluated for programmability, productivity and performance. The application will be tested on existing Tesla M1060 and latest "Fermi" (Tesla C2050) GPU architecture and the results compared.

The background and literature for this project is discussed in Chapter 2. Chapter 3 discusses the analysis and design for the optimization of the application. The optimization performed and the results obtained by using CUDAC programming model are described in Chapter 4. PGI Accelerator directives are used for accelerating the application and are discussed in Chapter 5. Chapter 6 discusses the results obtained and recommends the best practices to be used for accelerating applications using GPU.
Chapter 2

Background and Literature Review

In this chapter, the background and literature for the dissertation will be reviewed. The project aims to accelerate and optimize a Particle Physics application using Graphics Processing Units (GPU). The Physics of the application will be introduced and the architecture and programming models for GPU will be discussed in this chapter.

2.1 Particle Physics

2.1.1 Theoretical Particle Physics and Standard Model

Particle Physics is the study of the fundamental particle of matter and their interactions [1]. We currently believe that there are many types of particles namely leptons, quarks, gauge boson and Higgs boson. There are 6 quarks and 6 Leptons. Higgs boson contributes to the mass of the particle and all other particles have zero mass. These particles have strong, weak and electromagnetic interactions between them. The gravitational force is usually very small and is neglected.

The Standard Model explains most of the properties and interactions among particles. The electromagnetic interactions are due to exchange of particles called gauge bosons. The gauge boson is also called "force carrier". The weak interaction happens through bosons and strong interactions are carried by gluons. Force due to Gravity is not part of this model. Standard Model is considered to be the most accurate model for particle physics. This model doesn’t offer explanation for many cosmological findings like dark energy or dark matter in space. Though the Standard Model explains most of the interactions, many physicists believe the existence of particles in higher energy. Also this model doesn't account for the mass of the particles. This is an area of current research and there are many theories present that explain the same.

Peter Higgs in 1960s postulated that Higgs Boson associated with a "Higgs Field"[2], is responsible for the mass of the particles. Any particle interacting with Higgs field is given a mass and particles that don't interact have no mass. The interactions are through the Higgs boson. This particle is yet to be observed in any of the experiments. Many experiments are done to identify the same.

In particle physics high energy is used to create new particles and explore their properties. Initially cosmic rays, which are high energy rays were used to study particles. Nowadays most of the experiments are done by studying the beam of particles emitted by accelerators.
Two beams of particles travelling in opposite direction are made to collide with each other (colliders) and the particles emitted from high energy collision is detected using detectors.

The largest collider is the Large Hadron Collider (LHC) at CERN [3]. Here, two counter circulating beams of particles are accelerated and made to collide with each other. The detectors identify the particles emitted, their energy and momentum. Detector usually contains multiple sub detectors for much type of particles. These detectors are controlled by electronic and computer devices.

### 2.1.2 Lattice QCD Theory

Quantum Chromo Dynamics (QCD) is the theory that describes the strong interactions in a Standard Model [1]. Experimental data supports this theory. QCD describes the interactions due to gauge bosons, also called gluons. The gluons have no mass and have no charge. This theory has been successful in determining particle behavior for large momentum transfers [4]. In such conditions the coupling constant is large and we can use perturbation theory to solve the theory. Here, Coupling constant describes the strength of the interaction.

For high energy scales, the perturbation theory fails as the coupling constant becomes unity. Lattice QCD is a non perturbative method of simulating the particle interactions in a space-time lattice. Here the space and time are discretized. This means that particles can only move in discretized space and time, and therefore not in conformance with the continuum theme. Discretization of Lattice allows the interaction to be numerically simulated and find the correlation between the experiments and the simulation.

This simulation is based on the Monte Carlo simulation method. The inputs to these simulations are the coupling constants and the masses of the quarks (up, down, strange, charm and bottom). These input factors can be tuned to find if the results of simulation and experiments match. The main goal of Lattice QCD is to prove that QCD completely explains the strong interactions [4].

### 2.1.3 Lattice Perturbation Theory and Vertex function Calculation

Lattice regularization also introduces discretization errors and requires re-normalization to reduce the errors [5]. The renormalization constants can be used to obtains the continuum prediction from the Lattice QCD simulations. "Lattice Perturbation Theory" is a pertubative method for determining the renormalization constants. This method is reliable and systematically improvable [5].

The aim of the HPC application selected is to calculate the vertex function used for deriving lattice Feynman rules [6]. Following diagram shows quark particle travelling with momentum $p$, emitting gluon of momentum $k$ and then reabsorbing a gluon of momentum $k$. Here $p$, $q$ and $k$ refers to the momentum of the particles, solid line represents the quarks, circled line represents the emitted gluon and the solid circle represents vertex.
The vertex function \( Y \) for a point \((k, p, q)\) is given by

\[
Y = \sum_n f_n \times \exp \frac{i}{2} (k \cdot V_{n}^{(1)} + p \cdot V_{n}^{(2)} + q \cdot V_{n}^{(3)})
\]  

Here subscript \( n \) refers to the number of terms (nterms) in the calculation which can range from 17 to 30000. Each term in this sum is called “monomials” [6] and will be referred as “nterm” also. The monomial is defined by “\( f \)” its amplitude (complex) and \( V_{n}^{(1)}, V_{n}^{(2)} \) and \( V_{n}^{(3)} \) 4-component location vectors (integer). The combination of \((k, p, q)\) represents a “point”. The vectors \( k, p \) and \( q \) are complex and have 4 components each.

For complete derivation of the function please refer to the paper “Automated generation of lattice QCD Feynman rules” [6]. In this application vertex function is calculated for points. Number of points can be up to 100000. In the test application, \( k, p, q, f \) and \( V \) vectors are populated with random values and the vertex function and its derivatives are calculated for each point. The output is a vector of vertex function for each point and contains complex numbers.

2.2 GPGPU

General-purpose programming on graphics processing units (GPGPU)[7]are used to accelerate non-graphics applications using GPUs. Driven by the gaming market, these devices have evolved faster. These devices can be used for scientific computing also. These devices contain many cores and resemble Single Instruction Multiple Data (SIMD) architecture. They are used as an accelerator for the CPU (host) in a heterogeneous architecture. In the following sections GPU will be referred to as device and CPU as host.

CPUs have shown performance gain in the past due to increase in clock speed, transistor density and capacity. CPUs are optimized for single thread performance [8] using pipe-lining, caching, out of order execution and branch prediction. This means that much of the circuit is not dedicated for actual computing. This results in lesser floating point operations per second. Also increase in the clock rate has increased the heat dissipation and has made manufacturing CPUs costlier. This has resulted in using multiple cores in a single chip which share main memory and network bandwidth.

Though the CPU processing power is increasing, the memory has not become faster and has become a bottleneck for peak performance. Memory latency and bandwidth are important for most application performance. Though dual core processors have shows improved performance, Quad core and eight-core processors have shown lesser improvements for many large HPC applications that uses message passing programming model.
GPUs are optimized for 2D graphics image processing. Image processing is performed parallely by each thread working on an image pixel. The image pixel is represented as single precision floating point number. Many scientific application use similar algorithms and design. Such applications can be accelerated using GPU. GPUs have lower clock speed compared to CPU but high memory throughput and computational power. The lesser clock speed means that they dissipate lesser heat than the CPUs.

In GPU, most of the circuitry is dedicated for computation and can be used to accelerate application performance. The GPU is connected to the CPU through a network bus and acts like a heterogeneous device. GPU and CPU don't share any common memory and any data needs to be communicated. Communication of data between CPU and GPU is through the network bus. GPU has high computational power as they have more units dedicated for computation. The hardware thread management enables faster context switching of threads waiting for memory access and improves the memory throughput by hiding the memory latency.

NVIDIA's Tesla M1060 [9] and Tesla C2050 “Fermi” [10] AMD's Firestream [11] processors are widely used GPU for computing. Both the NVIDIA and AMD processors provide competing processing power but NVIDIA has better support for Double precision and Error Correcting Codes (ECC). In the following sections, GPU will refer to NVIDIA's Tesla or Fermi units unless specified otherwise.

2.2.1 GPU Architecture

NVIDIA's Tesla processors offer computing solutions in the form of GPGPUs. GPU (device) consists of hundreds of streaming processor cores. They are usually grouped together as Symmetric Multiprocessors (SM) as shown in Figure 2. Each SM consist many Scalar Processors (SP). Each SP is capable of performing Integer and Floating point arithmetic (in single and double precision). SM consists of load and store units for memory operations and special function units for sin, cos, exp functions. The Dispatch unit dispatches the instructions and the Warp Scheduler schedules the threads on the cores of the SM.

The SMs in GPU resemble MIMD architecture and can execute different functionality independently. The SMs cannot synchronize with each other but the share a global memory. The SPs within the SM resemble SIMD architecture and executes in Single Instruction Multiple Thread (SIMT) fashion [12]. They can synchronize execution and can communicate with each other using the shared memory.

Thread scheduling is managed in the hardware and enables multiple concurrent threads to be executed at the same time. Shared memory is available for each SM and enables faster memory access. Each device has a large Global memory that all cores can access and has a high memory bandwidth. Each core also has a local memory and set of registers. Texture memory is cached and enables texture fetching of data that has 2D or 3D locality. Constant memory provides faster read access to constant data and is also cached.

Tesla M1060 architecture (GT200) [9] is a multi-core architecture consisting of 240 scalar processors with a clock speed of 1.296 GHz. They are grouped together as 30 SMs with eight SP each. Each SM has 16 KB of shared memory and 16KB of registers. All SMs can access 4 GB of Global memory and provides a peak memory bandwidth of 102 GB/s. It provides peak performance of 78 GFlops for double precision arithmetic.
Figure 2 GPU Architecture showing Symmetric Multiprocessors (SM) and Scalar Processors (SP)

LD/ST – Load and Store Unit; SFU – Special Function Unit; FP – Floating point; INT – Integer; GPU – Graphic Processing Unit

[1] L2 Cache present in Tesla C2050 only

[2] L1 Cache present in Tesla C2050 only

Figure doesn’t show exact number of units.
Tesla C2050 (codename Fermi) \[10\] consists of 448 scalar processors with peak clock speed of 1.15 GHz. Each SM consists of 32 cores and they share 16-48 KB of shared memory which is configurable. It supports up to 6 GB of global memory and provides peak memory bandwidth of 144 GB/s. It provides peak performance of 515 Gigaflops for double precision arithmetic. Additionally C2050 provides Error Correcting Codes (ECC) for better reliability and two levels of cache, configurable L1 cache and unified L2 cache for better memory throughput.

The table below compares the features of the hardware used.

<table>
<thead>
<tr>
<th>Property</th>
<th>Tesla M1060</th>
<th>Tesla C2050 (Fermi)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Cores</td>
<td>240</td>
<td>448</td>
</tr>
<tr>
<td>Clock Speed</td>
<td>1.296 GHz</td>
<td>1.15 GHz</td>
</tr>
<tr>
<td>Number of SM</td>
<td>30</td>
<td>14</td>
</tr>
<tr>
<td>Number of SP per SM</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>Shared Memory per SM</td>
<td>16 KB</td>
<td>16-48 KB *</td>
</tr>
<tr>
<td>Register File size per SM</td>
<td>16 KB</td>
<td>32 KB</td>
</tr>
<tr>
<td>Peak Double precision performance</td>
<td>78 G Flops</td>
<td>515 G Flops</td>
</tr>
<tr>
<td>Peak Memory Bandwidth</td>
<td>102 GB/s</td>
<td>144 GB/s</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>NA</td>
<td>16 – 48 KB *</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>NA</td>
<td>768 KB</td>
</tr>
<tr>
<td>ECC Support</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Load/Store Units per SM</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>SFU Units per SM</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Warp Schedulers per SM</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Dispatch Unit per SM</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 1 Comparing Tesla M1060 and Tesla C2050 Hardware specifications

NA-Not available; * - configurable;

Both the above processors support asynchronous data transfer and hardware management of Threads. This allows thousands of threads to execute concurrently. The Fermi offers better double precision support. GT200 can issue only one double precision operation (fused multiply and add), whereas the Fermi can issue 16. It is evaluated that the Fermi shows 4.2x performance improvement over GT200 for double precision arithmetic. The dual scheduler can schedule two warps at a time parallely. Even if there are dual schedulers per SM, only one will be active for double precision operations.

The most important improvement in Tesla C2050 over M1060 is the increase in shared memory size from 16 KB to 64 KB. Also there is Hardware caching system in place (L1 cache) which manages caches for Global and Local memory accesses \[13\]. This make programming easier as the programmer can use this cache instead of managing the cache programmatically in the code. The L2 cache (768 KB) is
common for all SMs and this is also a new feature in this GPU Architecture. There are 16 load and store units which implies 16 threads can load and store parallely.

### 2.2.2 CUDA Programming Model

The CUDA (Compute Unified Device Architecture) [14] Programming model enables porting of applications to GPU. It encapsulates the hardware complexities and aims at providing an easier interface for the developers to interact with the device units. In this model, parallelism is achieved directly instead of using loops [14]. The unit of program that is executed on the GPU is called the Kernel. An application can have one or more kernels. This kernel is executed by many threads in parallel. This model is developed for image processing applications and can be compared to threads working on the pixels of an image in parallel.

The threads are grouped together as blocks. The thread block is scheduled to execute on any one of the SMs. They share the shared memory and registers of the SM. The threads in thread block are divided as thread warps which include 32 threads each. Thread warp represents the fundamental dispatch unit in a SM [14] and a SM can execute one or more thread warps in parallel. The thread blocks are again grouped together as Grids; a kernel executes a Grid of thread blocks. The threads and the blocks are identified by ids (threadid, blockid) in the kernel.

Each thread has a local memory and block of threads can also access the shared memory of the SM. All threads have access to global memory, texture memory and constant memory. The threads in a thread block can coordinate and synchronize among themselves. The model provides barrier synchronization for threads within the thread block but not across thread blocks. Any communication across blocks of threads can be achieved through global memory accesses.

This model scales by dividing the problem into many sub problems that can be executed by independent thread blocks [15]. Each sub-problem in turn is solved by the block of threads executing together. Kernels execute asynchronously in the GPU and the CPU needs to synchronize with the device to check for completion. This model also allows multiple kernels to be executed in the same device. Here the kernels are executed on one or more SMs.

The following Figure 3 shows the programming model and the memory hierarchy used. Here the threads and blocks are arranged in two dimensions for simplicity. Also the memory hierarchy shows the memory that is accessible at each level. This model also follows the processor hierarchy, where the Grid of blocks is scheduled on a GPU, block of threads on a SM and a thread on a single core.
2.2.3 CUDAC

CUDAC is an extension to C programming language and provides a simpler interface to CUDA device. Kernels are defined as C functions and the memory can be allocated and accessed using C variables. The interface also provides functions to synchronize thread execution, to allocate, deallocate memory (shared and global), transfer data between CPU and GPU etc. CUDAC is built on top of the low level CUDA Driver API. CUDA Driver API provides low level functionality but is harder to program.

The CUDAC files are named with ".cu" suffix. The ".cu" files are compiled with the nvcc compiler. The compiler separates host (CPU) code if any from the device code. The compiler converts the device code to a low level assembly code called the Parallel Thread eXecution (PTX) code. This PTX code can be executed on the GPU device using the CUDA Driver APIs. At runtime, the PTX code is again compiled to a binary code by the Driver API. This is called "just in time compilation" [14]. The separated host code is output as a C code that can be compiled by compilers.

The capability of the device architecture is expressed as compute capability (CC), represented by its major revision number and minor revision number. The major revision number denotes the core architecture and minor revision number, represents incremental changes to the core architecture. The latest Fermi architecture has 2.x compute capability whereas the Tesla M1060 has a compute capability of 1.3.
CUDAC is designed and developed only for NVIDIA devices and cannot be used for devices from other vendors. Thus CUDAC is not a portable solution for accelerating applications. But this is the most mature and stable programming language with good support for programming, profiling and debugging. NVIDIA has also developed many tools to improve productivity and analyze application performance.

2.2.4 PGI FORTRAN and C accelerator

PGI Fortran and C accelerator is a collection of directives that allows host code to be accelerated by a device [16]. These directives can be applied to a FORTRAN or C code and is portable across platforms and accelerators. This approach resembles the directive based OpenMP and is incremental. The directives allow specific regions of the host code to be offloaded to an attached accelerator. In case of GPUs, the compiler automatically generates the code for initializing the device, transferring data between host and device, executing the computation in the device.

Like OpenMP, the PGI runtime library provides APIs to access the devices properties. The compiler analyzes the code region and determines the memory that needs to be allocated in the device, if the data has to be copied in or out of the device. If the compiler analysis is unable to determine this, the code generation fails. Developer can also help the compiler by specifying the data movement through clauses like copy, copyin, copyout, local etc. Loops can be parallelized by using for/do directives and also provides scheduling options just like OpenMP. If the scheduling options are not mentioned, the compiler automatically determines the best scheduling for the loop.

PGI supports two execution models for the current GPUs: “doall” parallel loop level and and inner synchronous (“seq”) loop level [16]. "doall" is applied for fully parallelizable loops where the iterations are completely independent. "Inner synchronous" is applied to loops that can be vectorized but requires synchronization. The "parallel" and "vector" clauses of the directive determines the execution model used. Also iterations can be executed sequentially using the "seq" clause.

2.2.5 PGI Accelerator Programming Model

The PGI Accelerator programming model resembles the OpenMP model. Here, directives are used to help the compiler, map the functionality to the accelerator hardware. There are three types of directives namely, compute region, data region and loop directive [17]. Compute region is a mandatory directive that contains all the loops that will be executed in the accelerator. The data required for the compute region is automatically copied to the accelerator at the start of the region and the output copied back to the host at the exit of the region. This is determined automatically by the compiler.

The data region contains many compute regions and data regions. This allows the programmer to allocate and copy data in the accelerator. This helps the programmer to better manage the data movement between host and device. The Loop directive maps loops directly to the accelerator and allows programmer to map the loop iterations to the threads and blocks indexes. This mapping is done by a part of the compiler called Planner. The sample region is shown below.
Data region 1
  Compute region 1
    Loop region 1
    Loop region 2
  End Compute region 1
  Data region 2
    Compute region 2
      Loop region 3
    End Compute region 2
  End Data region 2
  Compute region 3
    Loop region 4
  End Compute region 3
End Data region 1

The Planner [17] maps the loops to the accelerator by tiling the loops. It assigns the grid indices to the outermost loops and thread indices to the innermost loops. The loops are ordered to reduce memory access or make them strided to improve memory throughput (Coalescing). Also the length of the innermost loop is optimized to improve shared memory usage (memory cache).

![Figure 4 Code Translations in PGI Programming](image)
2.2.6 OpenCL

Open Computing Language (OpenCL) [18] is an open standard, parallel programming for heterogeneous computing using modern architectures. The standard abstracts the underlying hardware architecture and programming model and provides a portable solution for accelerating applications. OpenCL is developed and maintained by the Khronos group.

OpenCL platform model consists of one host and many computing devices [19]. OpenCL allows programmers to write portable solution for using heterogeneous platforms containing CPUs, GPUs and other hardware devices. The OpenCL programming model and memory model resembles that of CUDAC. Here threads maps to work-items and thread blocks to work group. Each work-item has access to private memory, work-groups share a local memory and all work-groups have access to global and constant memory.

2.2.7 CUDA FORTRAN

CUDA FORTRAN [20] is a set of extension to FORTRAN that implements the CUDA programming model. This provides functionality to write kernels that can be executed on the device, copy data between host and device and allocating memory on device. This is similar to CUDA C and provides similar functionality and support.

2.3 Others work

Many researchers have successfully ported applications to GPUs and have achieved good speedups. Seismic wave simulations to detect Earth quakes have been accelerated using GPU cluster [21]. This application uses non-blocking Message passing approach for communication between the cluster nodes. They have reported speedups of 20x and 12x when compared to 4 core and 8 core CPU cluster implementations respectively. The production version runs on a 48 node GPU cluster (Tesla S1070) and the CPU used is two quad-core Intel Xeon X5570 Nehalem processors operating at 2.93 GHz.

In Lattice QCD, calculation of Hopping matrix takes most of computation time. This kernel has been accelerated using GPU (NVIDIA 8800 GTX) [22]. Researchers are able to achieve a speedup of 8.3 over an SSE2 optimized version on 2.8 GHz Xeon CPU. This implementation uses fine grained parallelism for optimization.

Researchers at the Taiwan have successfully ported lattice QCD simulation to GPU cluster having 128 GPUs (Tesla S1070 and Nvidia GTX285) [23]. They have shown performance of the order of 15.3 Teraflops. They have also shown that GPUs offer better price/performance ratio and power consumption when compared to conventional supercomputers.

Researchers at Hungary have successfully ported conjugate gradient calculation of Lattice QCD to GPU [24]. They have achieved speedup of 5 on NVIDIA 7900 GTX model. They used OpenGL as the programming language and single precision for all computations.
Clark and fellow researchers have developed a library for performing calculations in lattice QCD on graphics processing units (GPUs) using NVIDIA's "C for CUDA" API. This is called QUDA [25]. The library has optimized kernels for many lattice QCD functions. These functions also support double, single and mixed precision for floating point operations.
Chapter 3

Analysis and Design

This chapter describes the analysis and design for the optimization of the Particle Physics application that is accelerated using GPU. The application is a FORTRAN application and is accelerated using CUDAC kernel. This application has already been ported by Dr. Alistair Hart. In this chapter, the design of this accelerated application is discussed and its performance analyzed. Then the optimizations that can be performed for GPU accelerated code and how this applies to the current application is discussed in further sections.

3.1 Current Design

The calculation vertex function $Y$, given by equation (a) is the part of the application has been ported to GPU. This part takes almost 70% of the total runtime of the main application. The test application is a scaled down version to enable easier porting and optimization. For easier integration, the interface is made similar with the main application which is completely written in FORTRAN. In this application $k$, $p$, $q$, $f$ and $V$ vectors are populated with random values and the vertex function and its derivatives are calculated for each point. The ratio of computation per memory load is very high for this application. The calculations of vertex functions are completely independent and don’t depend on the order in which they are calculated. All this shows that the application can be accelerated using GPU.

As a part of the preliminary analysis this application is successfully built on the Ness GPU Device "Tesla M1060". The application was tested using 500 points and 8000 integration terms (nterms or monomials). The GPU accelerated code runs at 19 seconds whereas the CPU version computes in 20 seconds. The time taken to copy the data between GPU and CPU is found to negligible. Following graph Figure 5 shows the speedup achieved using GPU for different number of points and integration terms (nterms or monomials). Speedup is calculated as the ratio of time taken by the CPU code to time taken by the GPU accelerated code.

As seen in the Figure 5, the application shows increase in speedup with increase in number of points. But there is no significant change in speedup with increase in nterms.
An initial analysis of the code shows that each thread works on a single point, computes the factor due to each monomials and reduces their sum. This means that for 500 points, 500 threads will be scheduled. If the recommended scheduling option of 256 threads per block is used, this results in just two blocks of threads scheduled for this kernel. This is a limitation on the occupancy of the device as only two SMs will be occupied for computation. This explains why the speedup increases as the number of points increases as the device becomes more occupied. This indicates that the current approach is not scalable for lesser number of points (<4000).

The code uses local memory for all computation and no shared memory is statically allocated. The code loads the global memory data to its local memory before computing the result. In the code, all global memory access expect the writing to result vector are read only. This means that copying the data from global memory to local memory is an extra overhead as both the local memory and the global memory are off-chip and slow. Moving the read-only global data to constant or texture memory can improve memory throughput as they are cached.

Occupancy and Local memory accesses appear to be the major bottleneck for performance. The time to copy data between the device and the host is found to be negligible when compared to the computational time. The application was profiled using CUDA Visual profiler (version 2.3.11) for 500 points and 8000 nterms. It shows that occupancy as the limiting factor and a Global memory throughput of 0 GB/s. Also the profile indicates no shared memory used by the kernel.
3.2 Optimization

The basic principles for optimizing applications that are accelerated by GPU are similar to that of CPU. The aim is to increase the instruction and memory throughput, reduce memory latency and maximize parallel execution [15]. Utilization of all the cores is a huge factor in application performance as the device has hundreds of cores. Also the best optimization depends on the application characteristics.

Optimization of Instruction is a basic optimization and the compiler should do most of the optimizations. Using instruction with higher throughput should be favoured. Memory throughput can be increased by using the faster memory and managing the memory caches efficiently. The device and the host can execute parallelly and also a efficient parallel decomposition and algorithm should enable parallel execution of all cores of the device [15].

In the below sections, the optimizations that can be performed for the application will be discussed. In the following sections devices with compute capability (CC) 1.3 and 2.0 will be taken into consideration and double precision arithmetic will be used for all computation unless specified. Differences if any in optimization for these different CCs will be stated explicitly.

3.2.1 Memory Optimization

Optimizing the application for memory throughput is important in application performance as most of the current application performance seems to be limited by this. GPU with its different memory structures and access patterns makes it difficult to manage and optimize. Memory can be allocated like arrays and these array elements are sequential in memory [15]. The memory can be addressed using memory pointers and the address space is 32 bit and 40 bit for devices with 1.3 and 2.0 compute capability.

Memory is transferred between the device and the host as they don't share a common memory. Data from device memory is read and written by the device for computation. For higher memory throughput, transfer of data between host and device memory, global memory read/written by device should be minimized [15]. The device should use the shared memory as a memory cache. The shared memory can be allocated and managed using the software. It is important to note that local memory accesses are as slow as global memory accesses.

OPT1 Host and Device Memory transfer

The peak device memory bandwidth is of the order of 141 GBps (NVIDIA GeForce GTX 280) whereas the network bus has a bandwidth of 8GBps (PCIe ×16 Gen2) [15]. This huge difference in memory bandwidth implies that the transfer of data between host and device should be minimized. Also the number of data transfers should be reduced by batching the transfers together.

The design should map most of the algorithm to the device and reduce transfer of intermediate data. This means that the serial part or the less parallel part of the
algorithm should also be executed in the device. Higher bandwidth can be achieved by using page locked (pinned) host memory. Also when page locked memory can be mapped to the device memory the data transfers are done implicitly. This is a scarce resource and should be used with caution. Using large amount of page locked memory can affect the host performance.

In the particle physics application, the vertex calculation is completely ported to the GPU. The input data is copied to the device and then the kernel is launched. On successful completion of the kernel, the output data is then copied back to the host. The input data array is larger than that of output data. The input data can be pinned (page locked) to optimize memory bandwidth.

**OPT2 Coalescing global memory access**

Global memory is the single largest memory store in the device. This memory is not cached and its memory throughput can be improved by coalescing the global memory accesses. Global memory read and writes by half warp (1.x) or full warp (2.0) of threads is said to be coalesced, if they meet certain requirements. Global memory accesses by thread warps are managed by 32- 64- or 128- byte memory transactions [15]. The memory throughput increases if lesser number of memory transactions is issued.

Devices with compute capability 1.0 and 1.1 had very strict requirements for coalescing; this has become simpler in 1.3 and 2.0. If the memory is assumed to be aligned as 16 bit and 32 bit words, coalescing can be achieved if the memory accesses by threads in a warp fall within the memory segment of size, 32 bytes for 8-bit words, 64 bytes for 16-bit words, or 128 bytes for 32- and 64-bit words. Here the accesses need not be aligned.

The device aims at reducing the memory transactions by finding the memory segment that the lowest active thread has requested for and checks if all other thread requests fall within this segment. In this process the device also aims to reduce the memory transaction size. For example if only upper 64 bytes of the 128 byte segment is used, the device will reduce the transaction size to 64 byte.

CUDA follows IEEE 754 arithmetic format. The Integer data occupies 4 bytes of memory; float occupies 8 bytes; double occupies 16 bytes. Complex number is a structure with real and imaginary parts represented in double precision. Each Complex data occupies 32 bytes of global memory and a half warp (16 threads) with each thread accessing a complex number will require at least four memory transaction (128 byte) to complete a coalesced global memory access. In the application, following data is allocated (Column major order) in the global memory.

Complex f(1:nterms) //amplitude
Integer yxv(0:NDIR-1, 1:order+1, 1:nterms) //monomials
Complex k(0:NDIR-1, 1:order+1, 1:npoints) //points
Complex route(1:order+1, 1:npoints) //route
Complex vtx(0:Maxnum, 0:0, 0:0, 1:npoints) //result

In the above declaration, order represents the number of gluon legs, NDIR represents the number of directions and Maxnum represents the number of Taylor series factors.
Though the yxv array that represents the monomials is declared integer, they contain data in the range from -999 to 999 only. This can also be represented as "short" data type. The code has linear access pattern to the arrays and this implies that if kth thread in a warp accesses kth entry of the array, then global memory accesses will be coalesced. For example if ith thread in a warp accesses element f(i) or yxv(i,order,nterm) or k(i,order,npoint) or route(i,npoint) or vtx(i,0,0,npoint) then it is coalesced.

OPT3 Shared Memory and Bank conflicts

Shared memory is shared by all the cores of SM and is faster than global or local memory. This memory is divided into memory banks that can be accessed parallely by the threads unless there is a bank conflict. Bank conflict occurs when more than one thread in a half warp (CC 1.3) access the same memory bank [15]. In such cases, the memory access is split into separate requests that are conflict free. Each separate request made reduces the overall memory throughput.

The shared memory is organized so that successive 32 bit words are assigned to successive memory banks. This implies that threads accessing successive 32 bit words from the shared memory have no bank conflict. For devices with compute capability 1.3, the number of banks in shared memory is 16 and the memory requests from a thread warp (32 threads) are split into two requests for each half warp (16 threads).

For compute capability 2.0, the number of banks is 32 and the requests from a thread warp are not split into half warps. This implies that in CC 1.3, the shared memory request between threads of the first and second half warp cannot have bank conflict, whereas these access requests from first and second half warps are not protected against conflicts in CC 2.0. The design of the application should maximize usage of shared memory. A common approach would be to move the data from global memory to shared memory perform the computations and then moves the results back to global memory.

The original application doesn't use shared memory. It loads the values from global memory to local memory, performs the computations and then stores the result back to global memory. The local memory is as slow as global memory and a sufficient performance improvement can be achieved by replacing the local memory accesses with shared memory access.

As shown above, complex is the most prominent data type used in the application. Each element of complex occupies 128 bits long and occupies 4 banks of shared memory. For devices with compute capability 1.3, Complex data type accesses by half warp have a four way bank conflict. If the complex is represented as separate arrays of doubles, then conflict can be reduced to half. Bank conflict can be completely avoided if the double is represented again by two integers. The CUDA C provides APIs to convert double to int and vice versa but this will make application unmanageable.

In compute capability 2.0, the 64 bit (double) and 128 bit (complex) access patterns are specifically handled to avoid bank conflicts [15]. 64 bit access pattern has a bank conflict only if more than one thread from each half warps access different addresses
in the same memory bank. 128 bit access pattern shows a two way bank conflict. Porting the application to 2.0 devices should show performance improvement as the bank conflicts are minimized.

**OPT4 Overlapping memory transfer and computation**

As discussed before, the data needs to be copied from host to device as they don’t share a common memory. By default, this memory copy is performed synchronously and the control returns to the host only after the memory is copied. This copy can be performed asynchronously but requires the host memory to be pinned. This allows computation to be performed in the host and device when the memory is transferred. Also some devices allow memory transfer and computation to overlap. This overlap functionality again requires the host memory to be pinned. Where the memory copy is a big overhead, a good design is to transfer the memory in chunks and perform the computation in stages.

In the original application, the time taken for memory transfer is almost negligible. Any performance improvement by overlapping device kernel execution and memory transfer will be very less. But the host can execute parallely with the device. A part of the computation can be executed in the host based on the performance of the device. For example if the device accelerated code shows a speedup of 5, then one-fifth of the computation can be performed in the host. This saves time for memory transfer of one-fifth of the total data.

**OPT5 Texture, Local and Constant memory**

Local memory is local to the thread and is not cached. Like the global memory, this memory is off-chip [14] and is as slow as the global memory. The local memory is used to store automatic variables that cannot be stored in the registers. Large data structures and arrays are usually stored in the local memory and this is decided by the compiler. The current application uses local memory for all computations and this should be minimized for better performance.

Texture memory is a read-only memory that is cached. This memory is optimized for memory accesses that shows 2D spatial locality [14]. Texture memory provides an ideal alternative to global memory accesses. The memory access pattern need not be coalesced for textures. In the current application, most of the input arrays are read-only and has 2D spatial locality in their access pattern. They can be moved to texture memory for increasing the memory throughput. Cuda sdk 2.3[14] doesn’t support texture fetches for double precision numbers. These numbers should be represented as int2 and converted to double after texture fetching. The sample code is shown below

```c
//definition of texture containing int2 values
texture<int2, 2, cudaReadModeElementType> texRef;
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc(32, 32, 0, 0, cudaChannelFormatKindSigned);
//read int2 numbers from texture
int2 v = tex2D(texRef, r, p);
//convert int2 to double
double d_route = __hiloint2double(v.y, v.x);
```
Constant memory is a cached read-only memory. Each device has 64KB of constant memory and provides high memory throughput. Similar to texture memory, some of the input data can be moved to constant memory for better memory throughput.

### 3.3 Thread Scheduling

Load imbalance is one of the important bottlenecks for parallel performance. As the device provides hundreds of cores, a good balance in utilization of all the cores will result in peak performance. This is measured as the occupancy of the device [15]. The device executes the threads as warps and swap them when they are stalled or waiting for some resource. This is how the device hides any latency and keeps the hardware mostly occupied.

Occupancy is defined as the ratio of number of active warps in a SM to the maximum number of active warps possible. Occupancy of a device depends on many parameters. The number of active warps is limited by the number of threads that can execute at any point of time. Occupancy is not the prominent factor in application's peak performance. "Occupancy" is a measure of how many warps are active on the SM relative to the maximum. This does not directly have to do with the number of blocks. On a M1060, the maximum number of active warps per SM is 24 (768 threads), so 100% occupancy can be achieved with 3 blocks of 256 threads each, 8 blocks of 96 threads each, and anything in between. 100% occupancy cannot be achieved with 1 block, because the maximum number of threads per block is only 512. Thus this term signifies the number of active warps and not directly if all the cores of the SM are occupied or not.

Blocks are mapped to SMs, and warps of 32 threads at a time are executed in parallel by the 8 cores. So as long as the number of threads in your block is a multiple of 32, all the cores will be busy and occupied. The thread wars go to a waiting state for global memory reads to complete or for all threads to synchronize. The hardware reduces this wait by context switching the warps. This explains the need for more than one warp of thread to be scheduled in a SM.

The resources used by a block of thread can also limit the occupancy. When a kernel is scheduled, all the resources are allocated to the SM and the warps of threads are executed on the SM. This means that a block of threads that uses all the 16 KB of shared memory will have only one block of threads active for a SM. If the block uses only 4 KB of memory, then 4 blocks of threads can be active in a SM and so on. Here active means that the warps of threads are ready to be scheduled and executed on the SM. There are many SMs and kernel scheduling should ensure that all the SMs are occupied.

**OPT6 Resource utilization**

The threads are scheduled as blocks of threads and blocks of threads are grouped as Grids. Each thread has access to 16 KB (1.3) or 512 KB (2.0) of local memory and this is usually not the limiting factor. All threads in a thread block share the registers in the SM. The SM has a register memory of 16 KB (1.3) or 32 KB (2.0) and the number of registers used by each thread in a thread block directly impacts the
occupancy. Similar to registers, the block of threads also share the shared memory. Each SM has 16 KB (1.3) or 48 KB (2.0) of shared memory.

The number of registers allocated for each thread is determined by the compiler. The developer can also restrict the number of registers used. More number of registers used by each thread reduces the number of threads that are ready to be executed (active). Also this is dependent on the number of threads scheduled in a block. For example if each thread uses 12 registers (1.3) , thread block with 128 threads has better occupancy than a block with 256 threads.

Shared memory per block of threads can be determined at compile time and is dependent on the shared memory declared in the kernel. Similar to registers, the amount of shared memory used per block will determine the number of blocks of thread that can be active in a SM. For example, if a block of thread uses 4 KB of total shared memory (16 KB for 1.3), this means that 4 blocks of threads can be active.

**OPT7 Grid and Block scheduling**

The threads are scheduled in the device as Blocks and Grids. Threads and blocks can be multi dimensional and this allows easier mapping of the kernels to target algorithm. The dimensions of the thread schedule don't affect the performance of the device. But the number of threads per block and number of blocks per grid play an important part in determining the occupancy of the device.

Each block of thread is scheduled to run on a SM. To keep all the SMs in a device occupied, the number of blocks scheduled should be equal to the total number of SMs. In practice, each SM should have more than one block to execute, so that it can swap the thread warps waiting for synchronization or memory access to complete. [15] This ensures the hardware is always busy and occupied.

The number of threads in a block impacts the resource utilization and thus impacts the occupancy as shown in the previous section. Scheduling maximum number of threads in a block is always not the right option. For example, if device hardware limits the number of threads to 768, scheduling a block of threads with 512 threads (maximum) will have lesser occupancy than scheduling 3 blocks with 256 threads each.

As illustrated in the previous sections careful selection of shared memory, registers and thread scheduling is required for getting 100% occupancy. This process requires experimentation as 100% occupancy is not a requirement for peak performance. A lesser occupancy can result in better performance if the resources are utilized efficiently. Shared memory and registers should be utilized to the maximum to increase the memory throughput and the thread scheduling should ensure the occupancy doesn't impact the performance.

The original application shows a very poor occupancy as only 2 blocks of threads are scheduled. The parallel decomposition used in the application needs to be modified to ensure more blocks are used. This means that the existing problem needs to split into finer sub-problems. The profile of the application shows that for blocks of 256 threads, neither shared memory or the registers impacts the occupancy. The application can be optimized to maximize the usage of the resources and scheduling can be experimented for peak performance.
3.4 Parallel Decomposition

The application currently uses decomposition where result of each point is computed by a thread. This decomposition will be referred to as initial decomposition. This problem needs to be decomposed into many sub-problems that can run in parallel. The decomposed sub-problems may be completely independent or may require synchronization. Ability to parallelize any application is paramount for porting to GPUs. In the GPU programming model, blocks of threads can be assumed to work on a single unit of work. This assumption is a good because the block of threads share a common memory for communication and can synchronize their parallel execution.

Communication and Synchronization are part of most parallel algorithms. Faster shared memory and hardware multi-threading makes communication and synchronization costs low within a block. Communication across sub problems (block of threads) is through global memory read and write. This operation is not atomic by nature and synchronization should be done by using the atomic APIs. This is costly and should be avoided if possible.

Device occupancy requires that at least two or more blocks of threads assigned to a SM. The decomposition should ensure that there are as many sub-problems to keep the device busy. Finer decomposition to smaller sub-problems should be preferred as it will require lesser resources like shared memory and register. Lesser resource utilization improves occupancy. Where occupancy is not the limiting factor, this resource can be used to optimize the application performance.

This current application is decomposed such each thread computes the vertex function for a point. As discussed earlier, this approach has low occupancy. An easier approach will be to increase the number of threads that computes for each point. The number of threads should be multiples of 16 for ease of programming. Also a block of thread can be used (Block decomposition) or a warp of thread can be used (2-D decomposition) for computation of a point. All these approach will require different resource utilization and thread scheduling. The actual Block and 2-D decomposition is described in detail in 4.3 and 4.5 respectively.

The problem can be decomposed such that each block or group of threads works on a monomial and computes the factors for all points. This approach would require global synchronization as all the factors needs to be added. Atomic function can be used for the global additions but this approach is not recommended as atomic functions are slow.

3.5 Instruction Optimization

GPU are designed for single precision performance and this should be preferred over double precision. Instructions throughput is measured as number of clock cycles to complete the instruction in a SM [15]. Add and multiply instructions have better throughput than division or reciprocal operations. In devices with compute capability 1.3, single precision arithmetic is eight times faster than double precision. This gap is bridged in 2.0 where the double precision arithmetic is as fast as single precision.

The current application requires only double precision to be used.
3.5.1 Fast Math library

GPUs have special hardware units for functions like sin, cos and exp is exposed as a fast math library. This provides a faster implementation but is less accurate. The runtime API also provides equivalent versions that are more accurate but slower. The application has instruction that computes sin, cos and exp. These instructions can be optimized using the fast math library.
Chapter 4

Optimization and Results

In this section, the steps followed to optimize the code will be discussed and the results studied. At each step correctness of the application is tested by comparing the result with that of the CPU version. These results will be compared and discussed in detail in the next chapter. In this chapter, different code versions will be referred as CUDAC-WX or PGI-WX. CUDAC and PGI refer to the CUDAC and PGI Accelerator directive programming model used in the code respectively. WX refers to various code versions where X is any number and W is a working version which is tested for correctness. Also all the analysis was performed by varying the number of point in the application, keeping the number of nterms constant at 8000 unless specified otherwise.

4.1 System Hardware

The optimization was performed on the following hardware “Ness”, “Daresbury” and “Fermi”. In the following sections “Ness”, “Daresbury” and “Fermi” will indicate references to these systems. The table below shows the hardware specifications and compiler versions used.

<table>
<thead>
<tr>
<th>Specification</th>
<th>Ness</th>
<th>Fermi</th>
<th>Daresbury</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel(R) Xeon® CPU E5504 @ 2.00GHz</td>
<td>Intel(R) Xeon(R) CPU X5650 @ 2.67GHz</td>
<td>Intel(R) Xeon® CPU E5540 @ 2.53GHz</td>
</tr>
<tr>
<td>GPU</td>
<td>Nvidia Tesla M1060</td>
<td>Nvidia Tesla C2050</td>
<td>Nvidia Tesla T10 Processor</td>
</tr>
<tr>
<td>Cuda Version</td>
<td>2.3</td>
<td>3.1</td>
<td>2.3</td>
</tr>
<tr>
<td>PGI Version</td>
<td>Not Available</td>
<td>10.6</td>
<td>10.1</td>
</tr>
</tbody>
</table>

Table 2 Table showing the system hardware of different systems used

For complete details of the hardware specification refer Appendix B.
4.2 Initial Analysis and Profile

The initial analysis (complete analysis in 3.1) of the code on “Ness” showed that the application has poor occupancy and doesn't use shared memory efficiently. The initial code (CUDAC-W1) was analyzed using the NVIDIA cuda profiler version 2.3.11 [26]. This profile analysis shows that only 2 SMs out of the available 30 SMs is occupied for 500 points. It just uses 184 bytes of the 16KB of available shared memory per SM. The Figure 6 shows the “Profiler Counter Plot” of the initial CUDA code.

![Figure 6 Profile counter plot of the initial code](image)

In the following sections, references will be made to the CUDA profiler and its profiler counters. The profiler provides many automatic plots to help analyze the performance. The profile information contains many profiler counters which count the occurrence of a particular event in a thread warp. For example, gld_32/64/128b counters are incremented by one for each 32/64/128 byte transaction issued for global memory loads by a warp of thread. There are a group of such profiler counters and complete details of each counter are available in Appendix A. Also for the subsequent subsections, profiling and optimization is done on “Ness” hardware unless mentioned otherwise.

Also the profiler application has options to auto analyze the occupancy, memory throughput and profile counters. The application computes occupancy as percentage ratio of number of active warps to the maximum number of active warps. The global memory throughput is measured as GBps and bank conflicts are represented as warp-serialize rate (a number). The man pages also mention that there are errors in the calculation of warp-serialize rate [26].
Speedup will be used an important factor to measure the performance of the application. This speedup is measured as follows:

\[
\text{Speedup} = \frac{\text{Time taken for the serial code in Host}}{\text{Time taken for the Device accelerated code}}
\]

The GPU has an initial initialization overhead for running any accelerated kernel. This time usually ranges between 1.3 to 2.3 seconds. This time is not considered for any of the speedup calculation unless otherwise specified. The time taken on the device includes the time to allocate memory and copy data across the host and device also.

The CUDA profiler of the initial code shows that local loads and stores take almost 40% of the profiler counts. As the local memory loads and stores are slow and have high memory latency, moving this memory access to shared memory should provide improved performance. Also the actual code loads the data from the global memory to local memory and then accesses the data. This is not required as moving data from global memory to local memory will not improve the memory latency.

![Profiler Counter Plot](image)

**Figure 7 Profile Counter Plot of code version with direct global memory accesses**

All the local memory accesses are removed and the data is read directly from the global memory in the modified code (CUDAC-W2). The profile of the modified code is shown in Figure 7. The above change doesn’t impact the application performance and the profile shows no appreciable changes. The Figure 16 shows the speedup of the code versions CUDAC-W1 and CUDAC-W2. They show similar speedup for any number of points. Till 4096 points, the speedup increases as the number of points is increased and then flattens out around 5. The increase in speedup is due to the improvement in occupancy as more number of thread blocks is scheduled.
In the current application, threads are decomposed such that vertex function for a single point is calculated by a single thread (Initial decomposition). Figure 8 shows the decomposition where 4 threads operate on 4 points. This decomposition doesn’t require synchronization among the threads as they perform completely independent computations. As discussed in the above section, this decomposition suffers from poor occupancy. The threads are represented in single dimension for this decomposition. The timeline shows that all the columns (computation of a point) execute in parallel.

4.3 2-D Decomposition

The initial decomposition can be changed to 2-D decomposition where many threads (half warp of threads) work on a single point. The following Figure 9 illustrates this decomposition. The timeline shows that computation of vertex function for a point is performed in parallel by 4 threads and the workload is shared. Here the threads are represented in 2-dimensions for ease of programmability.

At each time step, thread \((\text{thread}_{p.x}, \text{thread}_{q.y})\) computes the result for point \(p\) and nterm \(q\). This decomposition has a better occupancy when compared to the initial decomposition. For example, for 500 points, with half warp of threads working on a single point, 32 blocks of 256 threads each, are scheduled. Since the nterms computation for a point is shared among threads, it requires synchronization to perform reduction of all the nterm factors. This is because the reductions in global memory and shared memory are not atomic.
To ensure atomic add, the reduction of monomial factors for a single point is serialized. But the reduction for “chunk size” number of points in a block execute in parallel.

![Diagram](image)

**Figure 9 2-D Decomposition shows 4 threads computing for each point. Each thread is represented in 2 dimensions as (threadp.x, threadq.y)**

The initial code was modified to use 16 threads (half warp) for computation of single point (CUDAC-W3). Half warp of threads load 16 nterm vectors parallely and compute their factors. The warp completes when all the monomial factors are computed. For a block of 256 threads, this means that 16 points will be computed in parallel with each half warp assigned to a point. This increases the occupancy of the application as 32 blocks of 256 threads each are scheduled for 500 points.

In the above optimization, shared memory is used to store the point and monomial vectors to improve memory latency. For a thread block of 256 threads, the 16 half warp of threads loads 16 (chunk size) points and 16 nterms into shared memory and then computes the result factors in parallel. The thread access pattern is modified for coalescing of global memory accesses. Here, the number of points computed per thread block is called the chunk size of the block. The profile of the application is shown below.
The above profile shows that the local loads and stores have taken over the instructions. This is a normalized profile for a SM. Normalized profile is the profile for a single SM. As more number of threads (16) work on a single point, for a block of 256 threads, the number of instructions are reduced to that for 16 points. In the previous decomposition, instructions were issued for 256 points and thus each block had more instructions to execute. This explains the increase in local load and store share when compared to instructions.

For 480 points, the kernel schedules 30 blocks and shows a good speedup of around 5. The profile analysis shows that Global memory read throughput is 1.21 GB/s; write throughput is 0.06 GB/s. The code shows high warp serialize rate. This is due to bank conflicts in shared memory. The application shows a constant speedup for 500 or more points as shown in Figure 16. While the speedup achieved for 4000 points or less is appreciable, the initial and the optimized code show similar performance for large number of points (>4000). This indicates that this approach has increased the occupancy but memory latency has not been reduced by using shared memory.

The profile analysis shows that the occupancy of the above code is limited by the shared memory. In the above code version, around 10KB of shared memory is used for each block and this limited the number of active warps to just 16 per SM. The shared memory is used to store the point and monomial vectors. For a block with 256 threads (chunk size of 16), 16 points and nterms values are stored in shared memory. Decreasing the chunk size will reduce the share memory required per block and increase occupancy. This scheduling option was experimented. Though profiler shows mark improvement in occupancy for thread blocks of size 32, 64 and 128, the speedup achieved remains the same. This shows that occupancy (limited by shared memory) is not the limiting factor for performance for this version of code.
4.4 Memory Optimization

Changing the decomposition as detailed in previous section, improves the performance of the application when number of points is less than 4000. For number of points greater than 4000, both the initial and 2-D decomposition (CUDAC-W3) shows similar performance as shown in Figure 16. In this section, the memory optimizations performed to improve performance will be discussed.

4.4.1 Vertex Function and Derivatives

Analyzing the application performance shows that the problem contains two parts, finding the vertex function for a point and finding its Taylor Derivatives. The second part takes almost 90% of the overall time and optimizing the memory and decomposition to speed up this part of the kernel will increase the overall speedup. The application code is modified (CUDAC-W4) to use shared memory to store the intermediate results and this code shows an improved speedup for any number of points as shown in Figure 16.

4.4.2 Inconsistent Shared Memory

To better utilize the shared memory in derivative calculation, the point and nterm vectors are moved to texture/constant memory. This shows no appreciable performance improvement over global memory accesses. This may be because the global memory accesses are coalesced and they perform better than cached texture memory access. The version of the application that uses texture memory works fine for 480 points (30 blocks).

In the above code version, when the number of blocks increases to more than 30, there are errors in texture fetches and results in error. The reason for this error is not identified but when more than 30 thread blocks are scheduled, more than one thread block is active per SM. The same error occurs when global memory access is used instead. The bug was later resolved by adding a syncthreads routine as follows:

```cpp
for(int r=0;r<nlegs;r++){
    int2 rval = tex2D(texRef,r,p);
    double d_route = __hiloint2double(rval.y, rval.x);
    for(int mu=0;mu<NDIR;mu++){
        int v_ind = mu + NDIR*(r + nlegs*(i+threadIdx.y));
        //this syncthreads require
        //if more than 30 blocks are scheduled ...
        //reason unknown.
        __syncthreads();
        //read from global memory
        u = v[v_ind];
        //read and write to shared memory
        s_mnm[threadIdx.x][threadIdx.y].x -= s_k[threadIdx.x][r][mu].y*u;
        s_mnm[threadIdx.x][threadIdx.y].y += s_k[threadIdx.x][r][mu].x*u;
        diff_factor[mu].y += d_route*u;
    }
}
```
The code was stripped down to a simpler version to find the bug.

```c
for(int r=0;r<nlegs;r++){
    for(int mu=0;mu<NDIR;mu++){
        int v_ind = mu + NDIR*(r + nlegs*(i+col));
        //this syncthreads require
        //if more than 30 blocks are scheduled
        __syncthreads();
        //read and write directly to global memory
        int u = v[v_ind];
        res[col+sres_p*p].x += int2double(u);
    }
}
```

The bug is not reproducible in this version. The issue seems to be with the shared memory “s_k” used. When data is directly read from global memory, the application shows no such bugs. This shows that there are some inconsistencies in shared memory accesses as many warps from different thread blocks are active. Care should be taken to ensure shared memory read and write are synched.

### 4.4.3 Shared Memory and Bank Conflicts

The code was modified to read npoints and nterms vectors directly from the global memory and shared memory was used to reduce the local memory loads and store in derivative calculation (CUDAC-W5). Figure 11 shows the profile of the code.

![Profile Counter Plot of code that uses all Shared memory in Complex format](image)

Profile of the above version of code shows a high warp serialize rate of around 290.443. The shared memory contains complex numbers which occupies 128 bits of memory. This means a single complex number is stored in 4 shared memory banks as each successive 32 bit words are stored in successive banks. Thus half warp of threads accessing successive complex numbers at the same time results in a 4 way
bank conflict. This explains the reason for high warp serialize rate. The declared shared memory variables in the application are shown below.

```c
__shared__ complex s_k[CHUNK_SIZE][MAX_NLEGS*NDIR+1];
__shared__ complex s_mmn[CHUNK_SIZE][CHUNK_SIZE+1];
__shared__ complex s_prod_pow_factor[CHUNK_SIZE][CHUNK_SIZE+1];
```

Just changing the final dimension to `CHUNK_SIZE+1`, decrease the warp serialize rate down to 79.248. For chunk size 16, `CHUNK_SIZE+1(17)` aligns the complex arrays to different banks to reduce bank conflicts. This version (CUDAC-W5) shows better speedup than previous versions with a peak speedup of 9.3 as shown in Figure 16. The slower speed up for lesser number of points (<4096) is attributed due to lower occupancy. For 4096 points, the device schedules 256 thread blocks and has peak performance for any more number of points/thread blocks.

The above code profile (Figure 11) shows a high warp serialize rate and it amounts to 50% of the counters. To reduce it, the complex number is split up into separate real and imaginary double arrays in the shared memory (CUDA-W6). This reduces the warp serialize rate to 87.94 as shown in the updated profile (Figure 12). This also increases the global memory read and write throughput to 4.37 GB/s. But this version shows similar speedup when compared to previous version (Figure 16). This shows that replacing complex number with separate real and imaginary doubles doesn’t help in application performance.

![Profiler Counter Plot](image)

**Figure 12 Profile Counter Plot of code that uses two double to represent a complex**

With double precision numbers occupying 8 bytes of memory, avoiding bank conflicts completely is impossible. So we will always have some bank conflict. One way to avoid bank conflicts is to split the double operands into two integer values as shown in the following code.
//int occupies 4 bytes of memory
__shared__ int shared_lo[32];
__shared__ int shared_hi[32];
//double occupies 8 bytes of memory
double dataIn;
//convert double to lo and hi integers
shared_lo[BaseIndex + tid] = __double2loint(dataIn);
shared_hi[BaseIndex + tid] = __double2hiint(dataIn);
//convert hi and low int to a double
double dataOut = __hiloint2double(shared_hi[BaseIndex + tid],
shared_lo[BaseIndex + tid]);

The cuda manual [14] recommends that this might not always improve performance
and does perform worse on devices of compute capabilities 2.0. This option can be
explored but ignored as it reduces the code readability.

The profile analysis shows that Occupancy is reduced to just 25% by using large
shared memories but this occupancy is enough to keep the SM busy for this given
problem as only the local loads and stores impact the performance. As discussed in
the design section (OPT6 Resource utilization), 100% occupancy is not limiting or
required for peak performance.

Figure 13 Profile Counter Plot showing better coalescing when thread index are
interchanged.

4.4.4 Other Memory Optimizations

As we have seen in design section (OPT5 Texture, Local and Constant memory), the
texture memory offers good alternative for global memory. This can be used to
replace shared memory if it is loaded only once and not reused or global memory
accesses if the access cannot be coalesced. The threadIdx.x and threadIdx.y variables
are automatic variable that denotes the thread index in 2 dimensions. Storing these variables in local variable and using the same impacts the performance.

As found earlier, coalescing global memory access doesn't impact the performance as the local loads and store take most of the time. In the current code, the global memory loads and stores are not fully coalesced. Even if there is no 64 or 128 byte memory transactions, the application shows no change in performance. Figure 13 shows the profile of the version of code where the thread index is interchanged for better coalescing. Even if this profile shows improvement in number of 128 and 64 byte transactions issued for global memory accesses, there is no improvement in speedup.

4.5 Block Decomposition

2-D decomposition offers good performance improvement over the initial decomposition. In 2-D decomposition, the reduction of nterm is serialized and requires synchronization. Using complete blocks of threads to work on a single point will increase the synchronization further. But this decomposition has lesser instructions per block of threads and reduction can be parallelized by using binary data structure. In the above method, each block works on 16 (chunk size) points. This decomposition has been changed to employ one point per block (Block Decomposition).

Figure 14 Block decomposition shows block of 4 threads working on a single point

Figure 14 shows this decomposition for a block of 4 threads. At each time step the four threads compute the factors due to 4 nterms parallelly. The reduction of all the nterm factors is done at the end and requires synchronization. The timeline shows that the computation of a point is performed in parallel.
This version of the code that uses Block decomposition (CUDAC-W7) shows a speedup of around 10 and runs at 2.23 seconds for 480 points. In this code, the whole atomic reduction of monomial factors has been parallelized using binary tree structure and shared memory. The following code shows addition of n numbers in log n steps using shared memory.

```c
for(int i=2;i<=CHUNK_SIZE;i=i*2){
    if(tx%i == 0){
        s_temp_x[tx] += s_temp_x[tx+i/2];
        s_temp_y[tx] += s_temp_y[tx+i/2];
    }
    __syncthreads();
}
```

This technique can also be applied to the 2-D decomposition also but will increase the synchronization overhead. This method scales well with increase in number of points and shows constant speedup even for lesser number of points as shown in Figure 16. Profile of the code (Figure 15) shows the local load and store have reduced to 24% and instructions, warp serialize counters are prominent.

**Figure 15 Profile Counter Plot of code that uses one block of thread for a point**

### 4.6 Other Optimizations

Using fast math library sinf, cosf, expf and setting the compiler option to use fast math library shows improvement in performance and the code runs at 2.16 seconds (for 480 points). But this has a error deviation of 0.000000232263 (e-6). This precision is not desirable for this particle physics application.

Part of the memory was made page locked to improve memory copy throughput across CPU and GPU. This resulted in many inconsistencies. The code throws invalid argument error and results in wrong results. The time to copy data across CPU and GPU for this application is found to be negligible and such measures will not result in any speedup.
Figure 16 Speedup on Ness (compared to host) for different CUDAC Code versions (nterms = 8000).

**CUDAC-W1**: Initial version (before optimization)
**CUDAC-W2**: Local memory accesses modified to access global memory directly
**CUDAC-W3**: Uses 2-D decomposition and shared memory to store point and monomial arrays
**CUDAC-W4**: Uses 2-D decomposition and shared memory to store point, monomial and shared memory for derivative calculation.
**CUDAC-W5**: Uses 2-D decomposition and shared memory for derivative calculation.
**CUDAC-W6**: Uses 2-D decomposition and shared memory for derivative calculation. Complex number represented as separate real and imaginary values.
**CUDAC-W7**: Uses Block decomposition and shared memory for derivative calculation. Complex number represented as separate real and imaginary values.

### 4.7 Fermi Optimization and Analysis

The optimization and profiling done on the “Ness” Hardware was discussed in previous sections. “Fermi” hardware has many enhancements over the “Ness” hardware as discussed in 2.2.1. This hardware was made available towards the end of the project and analysis was done on optimizing application for this hardware. The application should show better performance in “Fermi” as compared to “Ness”.

The final CUDA accelerated code that uses Block decomposition (CUDAC-W7 or CUDAC/Block-D) is tested on Fermi. It runs at 12.68 seconds for 4800 points. This is a speedup of around 2 over the “Ness” hardware. This accelerated code that uses
block decomposition, uses shared memory to reduce local memory accesses and optimize performance. This initial investigation shows that “Fermi” has performance improvement over the “Ness”.

The initial version of the CUDA code (CUDAC-W2 or CUDAC/Initial) that heavily uses local memory for data accesses runs at 12.76 seconds for 4800 points. This code uses initial decomposition where vertex function of a point is computed by a single thread, doesn’t uses any statically allocated shared memory. For 4800 points, the kernel schedules only 19 blocks. This is a marked improvement from the “Ness” hardware as this version doesn’t use any shared memory. This performance improvement can be explained as the Fermi has only 14 SMs (19 blocks keep the device occupied) and manages the L1 and L2 cache for local memory and global memory accesses.

The cache memory in Fermi can be configured to be used for L1 cache or shared memory. Using the –dlcm compilation flag, they can be configured at compile time to be cached in both L1 and L2 (-Xptxas -dlcm=ca) or in L2 only (-Xptxas -dlcm=cg). The following API provides programming support to configure the same before the kernel “MyKernel” is launched.

```c
//Runtime API :
//cudaFuncCachePreferShared: shared memory is 48 KB
//cudaFuncCachePreferL1: shared memory is 16 KB
//cudaFuncCachePreferNone: no preference

cudaFuncSetCacheConfig(MyKernel, cudaFuncCachePreferShared)
```

When no preference is set, shared memory is used as the preferred cache. Also the code should be compiled with flag -arch sm_20 to use Fermi architecture specific optimizations.

### 4.7.1 Shared Memory and Block Decomposition

The version of the code that uses Block decomposition (CUDAC/Block-D) was tested by configuring the cache memory of Fermi. For 4800 points, setting shared memory to 48 KB (16KB of L1 cache), runs at 12.67 seconds and setting the shared memory to only 16 KB (48KB of L1 cache), and runs at 22.65 seconds. This is a significant difference in speedup.

In this code, around 15 KB of shared memory is statically declared and setting the total shared memory to 48 KB increases the occupancy of the device. Setting shared memory to 16 KB and giving preference to L1 cache doesn’t yield better performance as occupancy is reduced. The main issue in application performance is the local memory accesses. In block decomposition, most of them have been moved to shared memory accesses and this code version performs better with increased shared memory. Figure 17 shows constant speedup of 10 for the code that uses Block decomposition and 48 KB of cache allocated for shared memory per block.

Comparing this performance with the Tesla performance, it can be seen that Tesla performs as good as the Fermi when the shared memory is limited to 16 KB. Thus the
improvement in code performance is simply down to more shared memory. This is surprising as the Fermi has more cores and better double precision support.

4.7.2 L1 cache and Initial Decomposition

The initial version of the code (CUDAC/Initial) that doesn’t allocate any static shared memory is tested on the Fermi. For 4800 points, setting shared memory to 48 KB (16KB L1 cache), runs at 12.76 seconds and setting the shared memory to 16 KB (48KB L1 cache), and runs at 9.79 seconds. As the code doesn't use any statically allocated shared memory, allocating more shared memory doesn’t help the application performance. The hardware seems to manage the cache better than the code and shows peak performance when 48 KB is allocated for L1 cache.

Fermi hardware manages cache for global and local memory accesses and optimization will require utilizing this to maximum effect. Figure 17 show that the initial decomposition that uses L1 cache for optimization shows a peak speedup of 16 for 8192 points but shows similar speedup for that of block decomposition as number of points is increased. The peak performance is due to better cache usage for 8192 points. This version shows lesser speedup for lower number of points (<4096) as the device is not occupied with enough thread blocks.

The version of the code that uses 2-D decomposition (CUDAC/2-DI) shows poor speedup when compared to other two decompositions as shown in Figure 17. The reason for this performance couldn’t be reasoned easily. With the experience of optimizing code in Tesla M1060 hardware using shared memory, the initial code was modified to use shared memory to store the monomial array. This version of code ran at 10.13 seconds when L1 cache is enabled and 13.22 seconds when shared memory is enabled. The performance degradation can only be attributed to the syncthreads routine called to synchronize all threads. This shows that synchronization of threads has greater impact over performance of Fermi when compared to Ness. Assuming all other factors don't contribute the version of code with lesser synchronization overhead will have better performance. Current application performance shows that it’s better to leave the caching to the hardware and concentrate on other optimizations.

4.7.3 Concurrent execution

Based on the speedup achieved, part of the computation can scheduled on the GPU and rest of the computation performed on the CPU concurrently. With the current speedup of around 10 on Fermi hardware, this approach doesn't yield any performance improvement. This approach will be significant only when GPU is used to accelerate a multi-core CPU that has a comparable performance. Also this requires that the GPU to be 100% occupied when the computations are shared.
4.7.4 ComputeProf

ComputeProf is a profiler tool from Nvidia [27]. It supports profiling and analyzing the application performance on Fermi. This is similar to the Cuda profiling tool used for Ness. This tool seems to crash and has bugs in analyzing the profile. Similar bugs for crashes in the tool were reported in developer’s forum. These issues made profiling and analyzing code on Fermi harder.

4.8 Spin and Color changes

The current version of the application assumes that all the monomials are same and they contribute to the vertex function of a point. A change in the Particle Physics theory assumes that the monomials have spin and colour and the vertex function should be reduced accordingly for each spin and colour pair values. This section discusses the changes to the application design and implementation for the spin and color changes. It is assumed that color values can range from 0 to 2 and spin values from 0 to 15. This is a fair assumption for this theory.
The direct result of this change is that the spin and color information has to be additionally passed to the device kernel and the vertex function vector with spin and colour components should be copied back to the host. Adding spin and color component to the result vector increases the memory size of the array by a factor of 48. This limits the kernel to 6000 or lesser points as there is not enough global memory to store the result vector.

The current application kernel uses local memory for reduction of all the nterm factors and then copies the same to the global memory at the end. Such approach cannot be followed as there is not enough local memory to store spin and color components. This code is changed to directly reduce the factors in global memory. Figure 18 and Figure 19 shows the speedup of this version of code (CUDAC-SC1) on Fermi and Ness hardware respectively. This is significantly lower than that shown by the initial version of the code without spin color changes (CUDAC/Initial).

The reason for the poor speedup is due to the random access of global memory which reduces the efficiency of the L1 cache. In the initial version, the local memory accesses improves the data locality and cache reuse. The random spin color value for monomials make cache reuse almost impossible as global memory access of result vector is random. This can be improved if the monomials are ordered by spin and color values.

The application can be modified to use Block or 2-D decomposition. Since the global and shared memory accesses are not atomic, synchronization is required to ensure correctness of the application. For testing purposes, Block decomposition is used and atomic functions are used to synchronize the global memory reductions. Here monomials are assumed to be not ordered by spin and color values. CUDA doesn’t support atomic add for double precision numbers and this needs to be implemented as a workaround using the following function. This feature is not supported on Ness (CC 1.3).

```c
// Double precision atomics are not supported on any device, so we emulate with atomicCAS().

static __inline__ __device__ double atomicAdd(double *addr, double val)
{
    double old=*addr, assumed;
    do
    {
        assumed = old;
        old = __longlong_as_double(atomicCAS((unsigned long long int*)addr,
            __double_as_longlong(assumed),
            __double_as_longlong(val+assumed)));
    } while(__double_as_longlong(assumed)!=__double_as_longlong(old));
    return old;
}
```
This version of the code (CUDAC-SC2) shows very poor speedup on Fermi as shown in Figure 18. This is because of the extra overhead of atomic function.

Ordering of monomials by spin and color values helps in data locality. This also enables local memory to be used for reduction of nterm factors for each spin color combination. The code is modified to order the monomials by spin and color values and the kernel updated to use local memory for reduction of nterms factor. This modified code (CUDAC-SC3) shows improved speedup and this resembles that of the initial version as shown in Figure 18 for Fermi and Figure 19 for Ness.

Figure 18 Speedup on Fermi (compared to host) for application with Spin/Col changes (nterms = 8000).

- **CUDAC-Initial**: Initial version of the application
- **CUDAC-SC1**: Monomials are not ordered based on Spin and Color values and uses initial decomposition
- **CUDAC-SC2**: Monomials are not ordered based on Spin and Color values and uses Block decomposition, Atomic functions for reduction.
- **CUDAC-SC3**: Monomials are ordered based on Spin and Color values and uses initial decomposition
Figure 19 Speedup on Ness (compared to host) for application with Spin/Col changes (n terms = 8000)

- **CUDAC-Initial**: Initial version of the application
- **CUDAC-SC1**: Monomials are not ordered based on Spin and Color values and uses initial decomposition
- **CUDAC-SC3**: Monomials are ordered based on Spin and Color values and uses initial decomposition
Chapter 5

PGI Accelerator

The CUDA C programming model is a complex programming model that requires programmers to manage the initialization of the device, movement of data from host to device, scheduling of work threads and managing various memories. Apart from these complexities, this model is not portable across all device platforms and other architectures. Many efforts have been made by programmers to develop libraries and compilers that will make programming such devices simpler and portable.

The PGI Fortran and C Accelerator programming model is a directive based approach for accelerating applications using GPUs. It resembles the OpenMP approach. Here the user supplies directives to specify regions of code that should be accelerated by the device. The model is portable across operating systems and accelerators.

The particle physics application can be accelerated using this model. Effort was made to port the application to PGI Accelerator programming model and this chapter explains the usability and issues of this model. The latest version of PGI Accelerator version 10.6 was used for all this analysis. In the following sections, all the changes are done in the FORTRAN code and the CUDAC section of the code is not used. Correctness of the code is tested by validating the results from the accelerated region with that on the host. The latest licensed version of the compiler was available only on Fermi and analysis was only done on that hardware.

To compile the accelerated code, the compiler should be supplied with flags specifying the target architecture. Following flags were used by default unless specified otherwise.

```
"-ta=nvidia:keepgpu,cc20,time –Minfo"
```

"ta=nvidia” specifies that the target architecture is a NVIDIA GPU. “keepgpu” flag specifies the compiler to retain the .gpu file generated in the translation process. “cc20” flag specifies the compiler to generate code for compute compatibility 2.0 for Fermi (CC 2.0). “time” flag links the application with a profile library to collect timing information and “Minfo” flag specifies the compiler to emit information about the accelerator region. These compiler messages can be used to optimize the accelerated region.
The compiler messages help the programmer understand the limitation of the compiler in porting a region to accelerator. The compiler accelerates the loops by tiling them. Loop tiling involves adding many induction variables and these variables also appear in the compiler message. These extra messages confuse the developer. In the following messages, only p is declared in the code and other variables are variable generated by the compiler.

Accelerator restriction: induction variable live-out from loop: i$a18
Accelerator restriction: induction variable live-out from loop: .dY0026
Accelerator restriction: induction variable live-out from loop: i$b19
Accelerator restriction: induction variable live-out from loop: .dY0027
Accelerator restriction: induction variable live-out from loop: p
Accelerator restriction: induction variable live-out from loop: i$c20
Accelerator restriction: induction variable live-out from loop: .dY0028

The PGI documentation doesn’t describe the compiler messages and is incomplete. This makes porting and optimizing applications using accelerator directives, difficult.

5.1 Direct Approach

The FORTRAN code that generates the vertex function and its derivative has the following loop format as shown below.

```fortran
npoints_lp: DO p = 1,npoints
  nterms_lp: DO i = 1,nterms
    Compute vertex function
    Compute taylor derivatives
  ENDDO nterms_lp
ENDDO npoints_lp
```

The code is successfully compiled using pgfortran and directive is added to accelerate this region. This direct approach doesn’t seem to work as the compiler ignores the accelerated region and throws warning messages that “Function and procedures are not supported”. (Appendix C.1 contains the complete accelerated region and compiler messages). This is because the compiler doesn’t support complex arithmetic functions like SUM, CMPLX and EXP.

The compiler message also indicates that the accelerated region is ignored. For 480 points and 8000 nterms, this code executes in parallel in the CPU at 8.45 seconds. The same code when compiled using iFort compiler runs at 65 seconds. The performance improvement of pgfortran compiled code is attributed to PGI compiler auto parallelizing the code to execute on multiple cores of the CPU.

To validate the compiler, a simple matrix multiplication was build and accelerated using PGI Accelerator. The compiler was able to automatically determine the data to be copied in and copied out of the device (copyin and copyout) and execute the matrix multiplication kernel in the GPU.
5.2 Compiler Issues

Accelerating the actual FORTRAN code was found to be impossible as the existing compiler doesn't seem to support many features like intrinsic functions or procedures. Effort was made to build simpler application and accelerate using PGI directives. Many validations seem to be missed by the compiler developers. The FORTRAN code that is accelerated should have all variables explicitly declared. This is important for the Planner to generate code as the generated code has errors if any variable is not declared.

The PGI Accelerator doesn’t support complex arithmetic or functions like exp, sum or complx. This means all these functions should be replaced by arithmetic expressions. This is a limitation of the compiler in its current version. The compiler depends on the nested loop to generate the kernel and analyses the loops for any dependence. This means that the existing FORTRAN code should be rewritten to represents complex numbers as separate real and imaginary values and using explicit loops to perform reduction (instead using SUM intrinsic). This appears like a C Style coding and will be referred as “C-Style” code in following sections.

The compiler seems to have issues when subroutines are used with “Contains” keyword. The compiler has issues in determining the scope of variables defined in the subroutines. It throws the following error "PGF90-F-0000-Internal compiler error. Generating a negative offset into dimension of target array” when subroutines are used. As per the PGI support, these are assertions in the compiler itself and occur when the compiler enters an unexpected state. This restricts the usage of any subroutines and all logic is manually inlined in the main program.

5.3 Accelerator directives and clauses

The following code shows simple vertex function calculation ported using accelerated directives. The copyin and copyout clauses specify the data that needs to be copied to and from the device memory. “do parallel” clauses specify that that the loop iterations can be executed parallely. “do vector” clause specify that the loop to be tiled to enable better shared memory management.

64 !Sacc region copyin (k,yxv) copyout(vtx)
65 !Sacc do parallel
66 DO p = 1,npoints
67 !Sacc do vector(256)
68   DO i = 1,nterms
69     DO r = 1,order+1
70       DO mu = 0,NDIR-1
71         vtx(0,0,0,p) = vtx(0,0,0,p) + k(mu,r,p)*yxv(mu,r,i)
72       END DO
73     END DO
74   END DO
75 END DO
76 !Sacc end region
The compiler generated comments are as follows for the above code is as follows:

```
main:
64, Generating copyout(vtx(:,:,,:,:))
Generating copyin(k(:,:,::))
Generating copyin(yxv(:,:,::))
66, Loop is parallelizable
68, Loop carried dependence of 'real(vtx)' prevents parallelization
   Loop carried dependence of 'real(vtx)' prevents vectorization
   Loop carried backward dependence of 'imag(vtx)' prevents vectorization
69, Loop carried dependence of 'real(vtx)' prevents parallelization
   Loop carried dependence of 'real(vtx)' prevents vectorization
   Loop carried backward dependence of 'imag(vtx)' prevents vectorization
70, Complex loop carried dependence of 'real(vtx)' prevents parallelization
   Complex loop carried dependence of 'imag(vtx)' prevents parallelization
   Loop carried dependence of 'imag(vtx)' prevents parallelization
   Loop carried backward dependence of 'imag(vtx)' prevents vectorization
Inner sequential loop scheduled on accelerator
Accelerator kernel generated
66, !$acc do parallel
68, !$acc do seq
   Non-stride-1 accesses for array 'yxv'
69, !$acc do seq
70, !$acc do seq
```

The compiler generates the copyin and copyout clauses with the actual dimensions. The compiler can automatically generate copyin and copyout clauses. Managing the data move is useful when the same data is reused by multiple kernels or when intermediate data need not be copied back to the host. Loop (66) can be parallelizable as vertex function of each point is not dependent on other points as illustrated in the compiler message.

The loops at line numbers 68, 69 and 70 are not parallelizable as there is a loop carried dependence. This is a simple reduction that can be parallelized using a binary tree data structure, but the compiler seems to ignore this. The compiler documentation specifies that parallel reduction will be supported in future versions. As a result of this all the inner loops are executed sequentially as indicated by the “do seq” clauses added by the compiler.

For 480 points and 800 nterms, this kernel runs with the following configuration: device=0 grid=480 block=1. The outer parallel loop is mapped to 480 blocks in the Grid and inner "do seq" clause makes sure that only one thread is active in a block. The compiler tries to align data to the loop index such that the global memory accesses can be coalesced. The yxv array doesn’t fulfill this condition and compiler shows a message. These non-stride accesses of yxv will not be coalesced if 480 threads are scheduled in 1 or 2 blocks of 480 or 240 threads each respectively. This message is not relevant for the above scheduling has only one thread per block.
5.4 C-Style Code

C-Style code refers to the coding style where complex numbers are represented as separate real and imaginary values and explicit loops are used instead of using any FORTRAN intrinsic functions. The kernel analysis clearly indicates that we need at least two parallelizable loops that can be mapped to grids and blocks. Though the reduction operation in the loop can be parallelized using a binary tree structure, this is not supported in the current version. Since complex arithmetic is not supported, the complex numbers are represented as two double integers. With these issues in porting application, the above code is rewritten as C-style code (PGI-W1). Refer to Appendix for the actual accelerated region and compiler generated messages.

As seen, the C-style code suits the compiler better than FORTRAN. Most C like code can be easily ported whereas FORTRAN cannot be as array functions and complex arithmetic are not supported. The real and imaginary parts of complex numbers should be represented as separate arrays of double. The existing CUDA C code is a good reference and effort was made to write CUDA C code logic in FORTRAN and accelerate using PGI directives. This seems like reinventing the wheel and not productive. But with the current version of the PGI compiler, there is no easy option. Refer Appendix C.2 for the accelerated region and the corresponding compiler messages.

The above code version runs with the configuration grid: [16] block: [32] and runs at 16417295 microseconds on GPU and 21125747 microseconds on CPU. This version uses local memory to reduce global memory accesses. This speedup shows improvement over the CUDAC version that uses block decomposition (CUDAC/Block-D) for lower number of points (< 2048) but the speedup slows down as the number of points is increased as shown in Figure 20. The main issue seems to be the “loop dependence” of variable that prevents parallelization or vectorization of loops.

5.5 SUM Intrinsic and Memory access pattern

In all these versions, directives have not been added intentionally and they will be added only if required. This approach will be followed in the following sections also, as unused directives result in errors. Any additional clauses to cache the data seems to be ignored. The kernel clause is used to generate a single kernel for the entire do loop; else the compiler generates a separate kernel for each loop within the npoints loop. This reduces the synchronization required between host and device. Also the accelerated kernel uses initial decomposition as used by the CUDAC code. Using 2-D or block decomposition is difficult as the shared memory is managed by the compiler and offers lesser control over the decomposition techniques.

The above code uses C style coding style and Alistair Hart recommended the usage of SUM intrinsic for real numbers instead of using loops. The vertex function requires contribution of all monomial to be accumulated. Instead of using Global add, these values are stored in separate array variables p3r(1:nterms) and p3i(1:nterms) and reduced at the end using the SUM intrinsic. This allows parallelization of the reduction in the loop (PGI-W2). The initialization of the GPU was done explicitly and
instructions are optimized. Refer Appendix C.3 for the accelerated region and the corresponding compiler messages.

This code runs with the following configuration grid: [16] block: [32] and at 1611413 microseconds on GPU and 2404450 microseconds on CPU. This is a performance improvement over the previous version of the code. This version also shows similar speedup to that of the CUDAC/Block-D code as shown in Figure 20. Using the similar logic for all derivatives result in memory allocation errors as there is not enough local memory.

There are non-stride-1 accesses for result array in the code. This is fixed by interchanging the array rows and columns of the result array in the accelerated region and then copying the result back to the original array (PGI-W3). Refer Appendix C.4 for the accelerated region and the corresponding compiler messages. This version requires the result vector to be rearranged to that of FORTRAN format. This code performs better and runs at 9710232 microseconds on GPU and shows better speedup than CUDAC/Block-D version (Figure 20). The slower speed up for number of points < 2048 is due to the rearrangement of the result array that is done on the host.

![Figure 20 Speedup on Fermi (compared to the host) for different PGI accelerated code versions (nterms = 8000).](image)

**Figure 20 Speedup on Fermi (compared to the host) for different PGI accelerated code versions (nterms = 8000).**

*PGI-W1: PGI accelerated region that uses C-Style Code*

*PGI-W2: Using SUM Intrinsic for reductions of real numbers in PGI-W1*

*PGI-W3: Changing memory access pattern for coalescing global memory accesses in PGI-W2.*

*CUDAC/Block-D: CUDAC version of that uses Block Decomposition.*
Unrolling of loops by a factor of 4 shows no improvement in performance. In the above code changing the thread scheduling by adding vector clause results in the following error "PGF90--0000-Internal compiler error Generating a negative offset into dimension of target array ". The same error happens when the code is compiled using 10.1. This is a bug in the compiler.

The generated executable cannot be moved to any GPU machine and executed. The executable requires licensed version of libpgacc.so library. Thus this model is portable only on all platforms with licensed compilers. The profiler tool for Fermi is crashing and that makes understanding the PGI generated code hard. This model couldn’t be tested in “Ness” as licensed version of the compiler was not available.
Chapter 6

Discussion of Results

Results are obtained for this application by accelerating and optimizing the application using different GPU architectures and programming models. All these results will be compared and discussed in this chapter.

6.1 Sub-Problem

Parallelizing application requires the problem to be split up into smaller sub problems that can be solved parallely. In this application, the problem is decomposed based on the points. Here computation of vertex function for each point is a sub problem and the number of points represents the number of sub problems in the solution.

Figure 21 represents the speedup for Fermi and Ness GPU accelerated versions when compared to the respective Ness CPU version. The device initialization time of 1.8 seconds is added to the timing data for calculating the total speedup achieved. On the Fermi hardware, lack of profiling tools made it difficult to measure or analyze the performance. So the analysis of speedup achieved on Fermi hardware is purely based on the results and existing Fermi hardware specifications.

PGI (Fermi-PGI/Accelerator) fails to allocate enough device memory for number of points more than 20480. This denotes that the PGI allocates more memory than actually required. The PGI accelerator version shows the peak speedup of 19 for 2048 points. This version of code uses Initial decomposition and at 2048 points, even if the device is not fully occupied (8 blocks) and data cache is used efficiently and results in peak speedup. The PGI accelerator also uses separate double arrays to represent real and imaginary parts of the complex numbers. While a complex number will be stored as 128 consecutive bits of memory and the separate real and imaginary values are accessed as 64 bits of memory each. This results in better coalescing and has lesser bank conflicts as discussed in design section (OPT3 Shared Memory and Bank conflicts). This version of the PGI Accelerator compiler (10.6) has many bugs and cannot be used in production as detailed in 5.2.

The PGI compiler (Fermi-PGI/CPU) shows a speedup of around 5 by auto parallelizing the application using the 24 cores available. The exact detail of the procedure is not
fstudied but assuming that the compiler is well standardized, this serves as an indicator of how well the GPU devices perform.

![Figure 21 Speedup (compared to Ness CPU) on Fermi and Ness for increasing number of points (nterms = 8000).](image)

Fermi –Tesla C2050 (Solid lines); Ness –Tesla M1060 (Dotted lines);
CUDAC/Initial : initial CUDAC version.
CUDAC/2-D : CUDAC code that uses 2-D decomposition.
CUDAC/Block D : CUDAC code that uses Block decomposition.
PGI/Accelerator : PGI Accelerator version of the code on Fermi
PGI/CPU : PGI version of the code on Fermi that uses CPU cores (No GPU) to parallelize application (OpenMP like).

The CUDAC code that uses initial (Fermi-CUDAC/Initial) and Block decomposition (Fermi-CUDAC/Block D) shows constant speedup for number of points greater than
The peak speedup of ~23 is achieved for 8192 number of points using initial decomposition on the Fermi hardware. This is because the initial decomposition uses L1 cache to manage the memory caching and at 8192 points, cache is efficiently used. Also for 8192 points, 32 blocks of threads are scheduled which keeps the device completely occupied. For lesser number of points (< 8192), the speedup suffers as the device is not fully occupied.

In Block decomposition (Fermi-CUDAC/Block D) and 2-D decomposition (Fermi-CUDAC/2-D), shared memory is managed by the program and doesn’t depend on the hardware caching to manage cache memory. Both these versions show a gradual increase in speedup as the number of points is increased. This gradual increase is attributed to synchronization overhead. For lesser number of points, there is lesser number of thread warps that can be swapped when waiting for synchronization. As the number of points increase, more number of thread blocks is scheduled and thread warps waiting for synchronization are swapped with thread warps from other blocks to hide synchronization overheads.

In Fermi, 2-D decomposition shows lesser speedup when compared to Block decomposition. The reason for this difference is the algorithm used for reduction of nterm factors. In, 2-D decomposition the sum of nterm factor for a single point is serialized, whereas in block decomposition, the same reduction is parallelized using binary tree data structure. On Ness, The initial decomposition (Ness-CUDAC/Initial) shows the lowest speedup as the memory cache is not used and has poor occupancy. The 2-D (Ness-CUDAC/2-D) and Block (Ness-CUDAC/Block D) decomposition performs better than the initial decomposition and shows similar performance as the memory is cached using shared memory.

Surprisingly 2-D decomposition shows peak performance in Ness but performs poorly in Fermi. This is because of better double precision support in Fermi hardware. In Ness (CC 1.3), each SM has only one double precision unit [14], whereas double precision arithmetic is supported by all cores of SM. It takes 32 clock cycles to complete a double precision arithmetic in Ness, whereas Fermi takes only 2 clock cycles. When a warp of thread is scheduled in a SM for double precision arithmetic, this is automatically serialized in Ness, whereas Fermi can execute them in parallel due to better support. Serialization of reduction of nterms factors in 2-D decomposition impacts the performance of “highly parallel” Fermi but doesn’t impact the “serial” Ness.

When the memory accesses are cached (Hardware or Software), the Fermi versions show almost a speedup of 2 over the Ness versions. This is attributed to the better double precision support in Fermi hardware. There is still scope of further optimization possible on the Fermi.

6.2 Problem size

As discussed in the previous section, sub problems determine how parallel the application is. Gustafson’s law suggest that a larger problem size is required to efficiently use a larger processor. The problem size determines peak performance as the systems become more powerful. For this application, the number of monomials
Figure 22 and Figure 23 represent the speedup for Fermi and Ness hardware for increasing number of monomials respectively.

Figure 22 Speedup on Fermi (compared to host) for increasing number of monomials (npoints = 4096).

- **CUDAC/Initial**: initial CUDAC version.
- **CUDAC/2-D**: CUDAC code that uses 2-D decomposition.
- **CUDAC/Block D**: CUDAC code that uses Block decomposition.
- **PGI/Accelerator**: PGI version of the code that uses accelerator directives.
- **PGI/CPU**: PGI version of the code that uses CPU cores (No GPU) to parallelize application (OpenMP like).

For Fermi, the CUDAC version using initial decomposition (CUDAC/Initial) shows almost constant speedup for any number of points. The same can be said about the PGI accelerated version (PGI/Accelerator) as it also uses initial decomposition. This is because this decomposition has no synchronizations as the threads work independently on each point.

The Block (CUDAC/Block D) and 2-D (CUDAC/2-D) decompositions suffer from synchronization and block scheduling overheads. For lesser number of monomials, there is lesser number of instructions and the thread synchronizations for warps of threads make the problem more serial. As the number of monomials increases the problem size increases and helps overcome the synchronization overhead.

Similar performance is seen in Ness hardware as shown in Figure 23. The difference is that the 2-D decomposition (CUDAC/2-D) performing better than Block...
decomposition (CUDAC/Block D) in Ness and vice versa in Fermi. The reason for this is discussed in previous section (6.1). The initial decomposition shows constant but lower speedup as expected as there is no synchronization. This indicates that the decomposition which doesn’t require synchronization of threads performs better if other factors are not considered.

![Graph showing speedup on Ness for increasing number of monomials](image)

**Figure 23 Speedup on Ness (compared to host) for increasing number of monomials (npoints = 4096)**

- **CUDAC/Initial**: initial CUDAC version.
- **CUDAC/2-D**: CUDAC code that uses 2-D decomposition.
- **CUDAC/Block D**: CUDAC code that uses Block decomposition.

### 6.3 Best Practices

Any application problem that can be split up into thousands of sub problems or that which has fewer sub-problems that can be parallelized further, are suitable for GPU acceleration. Memory latency, Occupancy and Synchronization determines the performance of most applications. Profiling the application helps the programmer concentrate on the exact optimization to be performed.

Recent Hardware improvements (Tesla C2050) in memory caching and double precision support have made programming and optimization easier. Since the Tesla C2050 hardware manages the cache (L1 and L2 cache), programmer should try to use this feature as much as possible instead of managing the cache programmatically. The cache should be configured to have more L1 cache memory or shared memory based
on the application memory usage. As a guideline an application kernel that allocates lesser than 10 KB of shared memory should prefer L1 cache for better performance.

If Tesla M1060 is used, the programmer should manage the shared memory to reduce memory latency. Most applications will benefit by using all available shared memory to reduce local and global memory loads. Though using all shared memory limits the occupancy of the application, this is not a limiting factor for peak performance.

The increased number of cores in GPU demands the accelerated problem to be highly parallel. Any application will have peak performance only if the device is completely occupied. The parallel decomposition of the problem determines occupancy. As a guideline, for a thread block with 256 threads, the number of thread blocks scheduled should be at least twice the number of SMs. Assuming the sub problems cannot be parallelized, for Tesla 2050 with 14 SMs to be fully occupied there should be at least \((14\times2\times256)\) 7168 sub problems. If the sub problems can be parallelized to use 256 threads, there should be at least 28 sub problems to keep a Tesla 2050 occupied.

Synchronization is required for communication among threads, atomic memory accesses and to ensure correctness of the solution. Atomic operations like reduction should be avoided as they have higher overhead and impacts the performance. The syncthreads routine should be used for synchronization and is faster. Parallel Algorithm and decomposition should be chosen so that synchronization among threads is reduced.

Directive based programming models seem to offer a good and direct solution for accelerating application using GPUs. But the current compilers are still under development and this approach cannot be used for production code.
Chapter 7

Conclusions

This dissertation analyzed the optimization of a GPU accelerated application on GPU hardware. Also programming models like CUDAC and Directive based are compared for programmability and performance. The accelerated application is highly parallel and is well suited for GPU acceleration. Approximately, a good speedup of 10 and 25 are achieved on Tesla M1060 and Tesla C2050 when compared to Intel CPU Xeon(R) CPU E5504 @ 2.00GHz. It is also found that PGI Accelerated compiler that uses a directive based approach for accelerating applications performs as good as the CUDAC versions.

Optimizing accelerated application requires careful study of the existing problem and the available hardware. The Particle Physics application derived most of its performance improvement from replacing the local memory loads and stores with that of memory cache (Hardware or Software). Scalability of the application is improved by modifying the parallel decomposition used. It is found that different decomposition algorithms are required to achieve peak performance for difference application settings (number of points and nterms). The restrictions in shared memory, local memory and registers available make application kernels difficult to extend (their functionality) or manage them. For example, a change in local memory usage (spin color changes) can easily change the application performance.

GPUs and CUDA programming model offers a complete solution for accelerating most applications. The tools support and documentation is found to be complete and very useful. Especially the profiler tool is very helpful in analyzing the application performance and optimizing the same. The documentation recommends most GPU settings and helps us in optimizing application performance. But still CUDA programming model is complex and difficult to optimize. The hardware advancements by NVDIA have improved the overall double precision performance and memory caching. The memory cache is managed by the hardware in Tesla C2050 and this makes managing the memory cache easier.

Directive based approach seems to be the future programming model for any accelerators as it offers a portable and productive solution. The current version of the compiler from PGI for accelerating GPUs using directives, suffers from many issues and limitations. With standardization of directive based approaches, and further
advancements in GPU hardware, GPUs will become a recommended accelerator for most applications in future.

7.1 Further Work

Tesla 2050 was available only towards the end of the project and there were issues in profiling the application due to profiler crashes. Nvidia plans to fix these issues in the near future and this should enable analysis of their performance and optimization of the application further. Though CUDAC and OpenCL programming models are widely used, directive based approach seems to be the future programming model for GPUs and other accelerator architectures.

PGI Accelerator is working on the next version of the compiler that supports complex arithmetic and many other features and bug fixes. Cray is in the process of developing an OpenMP like directive based approach for accelerating applications using GPUs. Analysing the translation process and performance of these compilers should help standardisation of directives.
# Appendix A

## Profiler Counter Details

<table>
<thead>
<tr>
<th>Profile Counter</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch</td>
<td>Number of branches taken by threads executing a kernel. This counter will be incremented by one if at least one thread in a warp takes the branch.</td>
</tr>
<tr>
<td>divergent branch</td>
<td>Number of divergent branches within a warp. This counter will be incremented by one if at least one thread in a warp diverges (that is, follows a different execution path) via a data dependent conditional branch. The counter will be incremented by one at each point of divergence in a warp.</td>
</tr>
<tr>
<td>instructions</td>
<td>Number of instructions executed</td>
</tr>
<tr>
<td>warp serialize</td>
<td>If two addresses of a memory request fall in the same memory bank, there is a bank conflict and the access has to be serialized. This counter gives the number of thread warps that serialize on address conflicts to either shared or constant memory.</td>
</tr>
<tr>
<td>sm cta launched</td>
<td>Number of threads blocks launched on a multiprocessor.</td>
</tr>
<tr>
<td>gld uncoalesced</td>
<td>Number of non-coalesced global memory loads. This counter is available only for GPUs with compute capability 1.1 or lower.</td>
</tr>
<tr>
<td>gld coalesced</td>
<td>Number of coalesced global memory loads</td>
</tr>
<tr>
<td>gld request</td>
<td>Number of global memory load requests. This counter is available only for GPUs with compute capability 1.2 or higher. On devices with compute capability 1.3 enabling this counter will result in increased counts for the &quot;instructions&quot; and &quot;branch&quot; counter values if they are also enabled in the same application run.</td>
</tr>
<tr>
<td>gld_32/64/128b</td>
<td>Number of 32 byte, 64 byte and 128 byte global memory load transactions. These increments by 1 for each 32, 64, or 128 byte transaction. These counters are available only for GPUs with compute capability 1.2 or higher.</td>
</tr>
<tr>
<td>gst uncoalesced</td>
<td>Number of non-coalesced global memory stores. This counter is available only for GPUs with compute capability 1.1 or lower.</td>
</tr>
<tr>
<td>gst coalesced</td>
<td>Number of coalesced global memory stores</td>
</tr>
<tr>
<td>gst request</td>
<td>Number of global memory store requests. This counter is available</td>
</tr>
</tbody>
</table>
only for GPUs with compute capability 1.2 or higher. On devices with compute capability 1.3 enabling this counter will result in increased counts for the "instructions" and "branch" counter values if they are also enabled in the same application run.

<table>
<thead>
<tr>
<th>gst 32/64/128b</th>
<th>Number of 32 byte, 64 byte and 128 byte global memory store transactions. These increments by 2 for each 32 byte transaction, by 4 for each 64 byte transaction and by 8 for each 128 byte transaction. These counters are available only for GPUs with compute capability 1.2 or higher.</th>
</tr>
</thead>
<tbody>
<tr>
<td>local load</td>
<td>Number of local memory loads</td>
</tr>
<tr>
<td>local store</td>
<td>Number of local memory stores</td>
</tr>
<tr>
<td>cta launched</td>
<td>Number of threads blocks launched on a TPC.</td>
</tr>
<tr>
<td>tlb hit</td>
<td>Number of instruction or constant memory cache hits.</td>
</tr>
<tr>
<td>tlb miss</td>
<td>Number of instruction or constant memory cache misses.</td>
</tr>
</tbody>
</table>

Table 3 Profile counters and description
### Appendix B

#### System Hardware Specification

<table>
<thead>
<tr>
<th></th>
<th>FERMI</th>
<th>DARESBURY</th>
<th>NESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA Driver Version</td>
<td>3.1</td>
<td>2.3</td>
<td>2.3</td>
</tr>
<tr>
<td>Device Name</td>
<td>Tesla C2050</td>
<td>Tesla T10 Processor</td>
<td>Tesla M1060</td>
</tr>
<tr>
<td>Device Revision Number</td>
<td>2</td>
<td>1.3</td>
<td>1.3</td>
</tr>
<tr>
<td>Global Memory Size</td>
<td>2817982464</td>
<td>4294770688</td>
<td>4294770688</td>
</tr>
<tr>
<td>Number of Multiprocessors</td>
<td>14</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>Number of Cores</td>
<td>448</td>
<td>240</td>
<td>240</td>
</tr>
<tr>
<td>Concurrent Copy and Execution</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Total Constant Memory</td>
<td>65536</td>
<td>65536</td>
<td>65536</td>
</tr>
<tr>
<td>Total Shared Memory per Block</td>
<td>49152</td>
<td>16384</td>
<td>16384</td>
</tr>
<tr>
<td>Registers per Block</td>
<td>32768</td>
<td>16384</td>
<td>16384</td>
</tr>
<tr>
<td>Warp Size</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Maximum Threads per Block</td>
<td>1024</td>
<td>512</td>
<td>512</td>
</tr>
<tr>
<td>Maximum Block Dimensions</td>
<td>1024, 1024, 64</td>
<td>512 x 512 x 64</td>
<td>512 x 512 x 64</td>
</tr>
<tr>
<td>Maximum Grid Dimensions</td>
<td>65535 x 65535 x 1</td>
<td>65535 x 65535 x 1</td>
<td>65535 x 65535 x 1</td>
</tr>
<tr>
<td>Maximum Memory Pitch</td>
<td>2147483647B</td>
<td>2147483647B</td>
<td>2147483647B</td>
</tr>
<tr>
<td>Texture Alignment</td>
<td>512B</td>
<td>256B</td>
<td>256B</td>
</tr>
<tr>
<td>Clock Rate</td>
<td>1147 MHz</td>
<td>1296 MHz</td>
<td>1296 MHz</td>
</tr>
<tr>
<td>Initialization time</td>
<td>7800 microseconds</td>
<td>3672458 microseconds</td>
<td>3672458 microseconds</td>
</tr>
<tr>
<td>Current free memory</td>
<td>2707947520</td>
<td>4254142208</td>
<td>4254142208</td>
</tr>
<tr>
<td>Upload time (4MB)</td>
<td>1357 microseconds (781 ms pinned)</td>
<td>1173 microseconds (749 ms pinned)</td>
<td>1173 microseconds (749 ms pinned)</td>
</tr>
<tr>
<td></td>
<td>Ness</td>
<td>Fermi</td>
<td>Daresbury</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------------</td>
<td>--------------</td>
<td>--------------</td>
</tr>
<tr>
<td>Download time</td>
<td>1234 ms</td>
<td>1084 ms</td>
<td>1084 ms</td>
</tr>
<tr>
<td></td>
<td>(714 ms pinned)</td>
<td>(738 ms pinned)</td>
<td>(738 ms pinned)</td>
</tr>
<tr>
<td>Upload bandwidth</td>
<td>3090 MB/sec</td>
<td>3575 MB/sec</td>
<td>3575 MB/sec</td>
</tr>
<tr>
<td></td>
<td>(5370 MB/sec pinned)</td>
<td>(5599 MB/sec pinned)</td>
<td>(5599 MB/sec pinned)</td>
</tr>
<tr>
<td>Download bandwidth</td>
<td>3398 MB/sec</td>
<td>3869 MB/sec</td>
<td>3869 MB/sec</td>
</tr>
<tr>
<td></td>
<td>(5874 MB/sec pinned)</td>
<td>(5683 MB/sec pinned)</td>
<td>(5683 MB/sec pinned)</td>
</tr>
</tbody>
</table>

Table 4 Hardware specification of GPUs in Ness, Fermi and Daresbury systems
Appendix C

PGI Accelerated region and compiler messages

C.1 Initial Code

Accelerated region:

```fortran
426 !$acc region
427 npoints_lp: DO p = 1, npoints
428 vtx(:, :, :, p) = 0
429
430 nterms_lp: DO i = 1, nterms
431 phase3 = CMPLX(0, 0.5d0) * SUM(
432   k(:, 1:nlegs, p) * yxv(:, 1:nlegs, i) )
433 ctemp = f(i) * EXP(phase3)
434
435 !$ acc temporary assignments
436 spin = 0
437 col = 0
438
439 !$ acc Set component 0: no derivatives
440 vtx(0, spin, col, p) = vtx(0, spin, col, p) + ctemp
441 ! We already did m=0 above, so won't need prod_pow_factor(0)
442 ! Actually prod_pow_factor does not need to be an array as we
443 ! only use entry m for each iteration of the loop
444 DO mu = 0, NDIR-1
445   diff_factor(mu) = CMPLX(0, 0.5d0) * &
446     SUM(route(1:nlegs, p) * yxv(mu, 1:nlegs, i))
447 ENDDO
448
449 pow_factor(:, 0) = 1
450 DO n = 1, tayl_order
451   pow_factor(:, n) = pow_factor(:, n-1) * diff_factor(:)
452 ENDDO
453
454 ! We already did m=0 above, so shouldn't include it
```

62
The PGI compiler messages:

hpsrc_gpu:
  172, Memory zero idiom, array assignment replaced by call to pgf90_mzero16
  176, maxval reduction inlined
vertex:
  426, Accelerator region ignored
  430, Accelerator restriction: function/procedure calls are not supported
  431, sum reduction inlined
  432, Accelerator restriction: function/procedure calls are not supported
  456, product reduction inlined
generate_multidx:
  494, Memory zero idiom, loop replaced by call to __c_mzero4
  513, sum reduction inlined

C.2 PGI-W1

Accelerate region:
122 !$acc region
123 !$acc do kernel private
diff_factor_imag, diff_factor_real, pow_factor_real, pow_factor_imag
, prod_pow_factor_real, prod_pow_factor_imag)
124         DO p = 1, npoints
125           DO i = 1, nterms
126             phase3real = 0
127             phase3imag = 0
128             DO mu = 0, NDIR-1
129               diff_factor_imag(mu) = 0
130               diff_factor_real(mu) = 0
131             END DO
132             END DO
133             DO r = 1, order+1
134             DO mu = 0, NDIR-1
135               phase3real = phase3real - kimag(mu, r, p) * yxv(mu, r, i)
136               phase3imag = phase3imag + kreal(mu, r, p) * yxv(mu, r, i)
137               diff_factor_imag(mu) = diff_factor_imag(mu) +
138                 route(r, p)*yxv(mu, r, i)
139             END DO
140             END DO
141             phase3real = 0.5 * phase3real
142             phase3imag = 0.5 * phase3imag
143             DO mu = 0, NDIR-1
diff_factor_imag(mu) = 0.5 * diff_factor_imag(mu)
diff_factor_real(mu) = 0.5 * diff_factor_real(mu)
END DO
ctempreal = exp(phase3real)*cos(phase3imag);
ctempimag = exp(phase3real)*sin(phase3imag);
phase3real = freal(i)*ctempreal - fimag(i)*ctempimag
phase3imag = freal(i)*ctempimag + fimag(i)*ctempreal
vtxgpureal(0,0,0,p) = vtxgpureal(0,0,0,p) +
                      phase3real
vtxgpuimag(0,0,0,p) = vtxgpuimag(0,0,0,p) +
                      phase3imag
DO mu = 0,NDIR-1
  pow_factor_real(mu,0) = 1
  pow_factor_imag(mu,0) = 0
  ctempreal = diff_factor_real(mu)
  ctempimag = diff_factor_imag(mu)
  DO n=1,taylor_order
    pow_factor_real(mu,n) =  pow_factor_real(mu,n-1)* ctempreal - pow_factor_imag(mu,n-1) * ctempimag
    pow_factor_imag(mu,n) =  pow_factor_imag(mu,n-1)* ctempreal + pow_factor_real(mu,n-1) * ctempimag
  END DO
END DO
END DO
DO m=1, taylor_maxidx(taylor_order)
  prod_pow_factor_real = 1
  prod_pow_factor_imag = 0
  DO mu=0,NDIR-1
    rtemp = prod_pow_factor_real
    ctempreal = pow_factor_real
               (mu, taylor_multidx(m,NDIR-1-mu))
    ctempimag = pow_factor_imag
                (mu, taylor_multidx(m,NDIR-1-mu))
    prod_pow_factor_real = prod_pow_factor_real
                         * ctempreal - prod_pow_factor_imag
                         * ctempimag
    prod_pow_factor_imag = rtemp * ctempimag
                          + prod_pow_factor_imag * ctempreal
  END DO
vtxgpureal(m,0,0,p) = vtxgpureal(m,0,0,p) +
                      prod_pow_factor_real * phase3real
                      - prod_pow_factor_imag * phase3imag
vtxgpuimag(m,0,0,p) = vtxgpuimag(m,0,0,p) +
                      prod_pow_factor_real * phase3imag
                      + prod_pow_factor_imag * phase3real
END DO
END DO
END DO
!$acc end region

PGI Compiler Messages:

122, Generating copyin(taylor_multidx(1:taylor_maxidx,0:ndir-1))
        Generating copy(vtxgpuimag(0:taylor_maxidx,0,0,1:npoints))
        Generating copy(vtxgpureal(0:taylor_maxidx,0,0,1:npoints))
Generating copyin(fimag(1:nterms))
Generating copyin(freal(1:nterms))
Generating copyin(yxv(0:ndir-1,1:order+1,1:nterms))
Generating copyin(kimag(0:ndir-1,1:order+1,1:npoints))
Generating copyin(kreal(0:ndir-1,1:order+1,1:npoints))
Generating copyin(route(1:order+1,1:npoints))
Generating copyin(taylor_maxidx(taylor_order))
Generating compute capability 2.0 binary
124, Loop is parallelizable
   Accelerator kernel generated
124, !$acc do parallel, vector(32)
   Non-stride-1 accesses for array 'vtxgpuimag'
   Non-stride-1 accesses for array 'vtxgpureal'
   CC 2.0 : 51 registers; 4 shared, 404 constant, 88 local memory bytes; 16
   occupancy
125, Loop carried dependence of 'vtxgpureal' prevents parallelization
   Complex loop carried dependence of 'vtxgpureal' prevents parallelization
   Loop carried backward dependence of 'vtxgpureal' prevents vectorization
   Loop carried dependence of 'vtxgpuimag' prevents parallelization
   Complex loop carried dependence of 'vtxgpuimag' prevents parallelization
   Loop carried backward dependence of 'vtxgpuimag' prevents vectorization
   Loop carried reuse of 'diff_factor_imag' prevents parallelization
   Loop carried reuse of 'diff_factor_real' prevents parallelization
   Loop carried dependence of 'pow_factor_real' prevents parallelization
   Loop carried backward dependence of 'pow_factor_real' prevents vectorization
   Loop carried dependence of 'pow_factor_imag' prevents parallelization
   Loop carried backward dependence of 'pow_factor_imag' prevents vectorization
   Loop carried dependence of 'pow_factor_imag' prevents vectorization
   Loop carried dependence of 'pow_factor_real' prevents vectorization
128, Loop is parallelizable
132, Loop carried reuse of 'diff_factor_imag' prevents parallelization
133, Loop is parallelizable
141, Loop is parallelizable
151, Loop is parallelizable
156, Loop carried dependence of 'pow_factor_real' prevents parallelization
   Loop carried backward dependence of 'pow_factor_real' prevents vectorization
   Loop carried dependence of 'pow_factor_imag' prevents parallelization
   Loop carried backward dependence of 'pow_factor_imag' prevents vectorization
162, Loop is parallelizable
165, Loop carried scalar dependence for 'prod_pow_factor_real' at line 166
   Loop carried scalar dependence for 'prod_pow_factor_real' at line 169
   Scalar last value needed after loop for 'prod_pow_factor_real' at line 172
   Scalar last value needed after loop for 'prod_pow_factor_real' at line 173
Loop carried scalar dependence for 'prod_pow_factor_imag' at line 169
Loop carried scalar dependence for 'prod_pow_factor_imag' at line 170
Scalar last value needed after loop for 'prod_pow_factor_imag' at line 172
Scalar last value needed after loop for 'prod_pow_factor_imag' at line 173

C.3 PGI-W2

Accelerated region:

162 !$acc region
163 !$acc do private (diff_factor_imag,pow_factor_real, &
164 !$acc pow_factor_imag, prod_pow_factor_real, prod_pow_factor_imag &
165 !$acc , p3r, p3i )
166 DO p = 1,npoints
167     p3r(:) = 0
168     p3i(:) = 0
169     DO i = 1,nterms
170         phase3real = -0.5d0*SUM(kimag(:,:,p)*yxv(:,:,i))
171         phase3imag =  0.5d0*SUM(kreal(:,:,p)*yxv(:,:,i))
172         DO mu = 0,Ndir-1
173             diff_factor_imag(mu) =
174               0.5d0*SUM(route(:,p)*yxv(mu,:,i))
175         ENDDO
ctempreal = exp(phase3real)*cos(phase3imag);
ctempimag = exp(phase3real)*sin(phase3imag);
phase3real = freal(i)*ctempreal - fimag(i)*ctempimag
phase3imag = freal(i)*ctempimag + fimag(i)*ctempreal
p3r(i) = phase3real
p3i(i) = phase3imag
181
182     DO mu = 0,NDIR-1
183         pow_factor_real(mu,0) = 1
184         pow_factor_imag(mu,0) = 0
185         ctempimag = diff_factor_imag(mu)
186     DO n=1,taylor_order
187         pow_factor_real(mu,n) =
188               - pow_factor_imag(mu,n-1) * ctempimag
189         pow_factor_imag(mu,n) =
190               + pow_factor_real(mu,n-1) * ctempimag
191     END DO
192 END DO
193
194 END DO
195
196 !We've already done m=0, so skip it here
197 DO m=1, taylor_maxidx(taylor_order)
198     prod_pow_factor_real = 1
199     prod_pow_factor_imag = 0
200     DO mu=0,NDIR-1
201         rtemp = prod_pow_factor_real
202         ctempreal = pow_factor_real &
203               (mu, taylor_multidx(m,NDIR-1-mu))
204         ctempimag = pow_factor_imag &
205               (mu, taylor_multidx(m,NDIR-1-mu))
206         prod_pow_factor_real = prod_pow_factor_real *
207                           ctempreal - prod_pow_factor_imag *
208                           ctempimag
209     prod_pow_factor_imag = rtemp * ctempimag
+ prod_pow_factor_imag * ctempreal
201                       END DO
202                 vtxgpureal(m,0,0,p) = vtxgpureal(m,0,0,p)
+ prod_pow_factor_real * phase3real
- prod_pow_factor_imag * phase3imag
203                 vtxgpuimag(m,0,0,p) = vtxgpuimag(m,0,0,p)
+ prod_pow_factor_real * phase3imag
+ prod_pow_factor_imag * phase3real
204                    END DO
205             !End of derivatives section
206                    END DO
207                   !End loop over monomials
208     vtxgpureal(0,0,0,p) = SUM(p3r)
209     vtxgpuimag(0,0,0,p) = SUM(p3i)
210                END DO
211             !End loop over points
212           !$acc end region

PGI Compiler Messages:
162, Generating copyin(taylor_multidx(1:taylor_maxidx,0:ndir-1))
    Generating copyin(vtxgpureal(1:taylor_maxidx,0,0,1:npoints))
    Generating copyout(vtxgpureal(0:taylor_maxidx,0,0,1:npoints))
    Generating copyin(vtxgpuimag(1:taylor_maxidx,0,0,1:npoints))
    Generating copyout(vtxgpuimag(0:taylor_maxidx,0,0,1:npoints))
    Generating copyin(fimag(1:nterms))
    Generating copyin(freal(1:nterms))
    Generating copyin(route(1:order+1,1:npoints))
    Generating copyin(kimag(0:ndir-1,1:order+1,1:npoints))
    Generating copyin(yxv(0:ndir-1,1:order+1,1:nterms))
    Generating copyin(kreal(0:ndir-1,1:order+1,1:npoints))
    Generating copyin(taylor_maxidx(taylor_order))
    Generating compute capability 2.0 binary
166, Loop is parallelizable
    Accelerator kernel generated
166, !$acc do parallel, vector(32)
    Non-stride-1 accesses for array 'vtxgpuimag'
    Non-stride-1 accesses for array 'vtxgpureal'
    CC 2.0 : 50 registers; 4 shared, 476 constant, 88 local memory bytes; 16
    occupancy
167, Loop is parallelizable
168, Loop is parallelizable
169, Loop carried dependence of 'diff_factor_imag' prevents parallelization
    Loop carried backward dependence of 'diff_factor_imag' prevents vectorization
    Loop carried dependence of 'pow_factor_real' prevents parallelization
    Loop carried backward dependence of 'pow_factor_real' prevents vectorization
    Loop carried dependence of 'pow_factor_imag' prevents parallelization
Loop carried backward dependence of 'pow_factor_imag' prevents vectorization
Loop carried dependence of 'pow_factor_imag' prevents vectorization
Loop carried dependence of 'pow_factor_real' prevents vectorization
Loop carried dependence of 'vtxgpureal' prevents parallelization
Loop carried backward dependence of 'vtxgpureal' prevents vectorization
Loop carried dependence of 'vtxgpuimag' prevents parallelization
Loop carried backward dependence of 'vtxgpuimag' prevents vectorization

170, sum reduction inlined
  Loop is parallelizable
171, sum reduction inlined
  Loop is parallelizable
172, Loop is parallelizable
173, sum reduction inlined
  Loop is parallelizable
182, Loop is parallelizable
186, Loop carried dependence of 'pow_factor_real' prevents parallelization
  Loop carried dependence of 'pow_factor_imag' prevents parallelization
  Loop carried backward dependence of 'pow_factor_imag' prevents vectorization
  Inner sequential loop scheduled on accelerator
192, Loop is parallelizable
195, Loop carried scalar dependence for 'prod_pow_factor_real' at line 196
  Loop carried scalar dependence for 'prod_pow_factor_real' at line 199
  Scalar last value needed after loop for 'prod_pow_factor_real' at line 202
  Scalar last value needed after loop for 'prod_pow_factor_real' at line 203
  Loop carried scalar dependence for 'prod_pow_factor_imag' at line 199
  Loop carried scalar dependence for 'prod_pow_factor_imag' at line 200
  Scalar last value needed after loop for 'prod_pow_factor_imag' at line 202
  Scalar last value needed after loop for 'prod_pow_factor_imag' at line 203
  Inner sequential loop scheduled on accelerator

208, sum reduction inlined
  Loop is parallelizable
209, sum reduction inlined
  Loop is parallelizable

C.4 PGI-W3

Accelerated region:
161 !$acc region
162 !$acc do private (diff_factor_imag,pow_factor_real, &
163 !$acc pow_factor_imag , prod_pow_factor_real,
        prod_pow_factor_imag &
164 !$acc , p3r, p3i )
165        DO p = 1,npoints
DO i = 1, nterms
  phase3real = -0.5d0*SUM(kimag(:,:,p)*yxv(:,:,i))
  phase3imag =  0.5d0*SUM(kreal(:,:,p)*yxv(:,:,i))
  DO mu = 0,Ndir-1
    diff_factor_imag(mu) = 0.5d0*SUM(route(:,p)*yxv(mu,:,i))
  ENDDO
  ctempreal = exp(phase3real)*cos(phase3imag);
  ctempimag = exp(phase3real)*sin(phase3imag);
  phase3real = freal(i)*ctempreal - fimag(i)*ctempimag
  phase3imag = freal(i)*ctempimag + fimag(i)*ctempreal
  p3r(i) = phase3real
  p3i(i) = phase3imag

  DO mu = 0,NDIR-1
    pow_factor_real(mu,0) = 1
    pow_factor_imag(mu,0) = 0
    ctempimag = diff_factor_imag(mu)
    DO n=1,taylor_order
      pow_factor_real(mu,n) = - pow_factor_imag(mu,n-1) * ctempimag
      pow_factor_imag(mu,n) =  + pow_factor_real(mu,n-1) * ctempimag
    END DO
  END DO
  !We've already done m=0, so skip it here
  DO m=1, taylor_maxidx(taylor_order)
    prod_pow_factor_real = 1
    prod_pow_factor_imag = 0
    DO mu=0,NDIR-1
      rtemp = prod_pow_factor_real
      ctempreal = pow_factor_real
      (mu, taylor_multidx(m,NDIR-1-mu))
      ctempimag = pow_factor_imag
      (mu, taylor_multidx(m,NDIR-1-mu))
      prod_pow_factor_real = prod_pow_factor_real
      * ctempreal - prod_pow_factor_imag
      * ctempimag
      prod_pow_factor_imag = rtemp * ctempimag
      + prod_pow_factor_imag * ctempreal
    END DO
  END DO
  vtxgpureal(p,0,0,m) = vtxgpureal(p,0,0,m)
  + prod_pow_factor_real * phase3real
  - prod_pow_factor_imag * phase3imag
  vtxgpuimag(p,0,0,m) = vtxgpuimag(p,0,0,m)
  + prod_pow_factor_real * phase3imag
  + prod_pow_factor_imag * phase3real
END DO
!End of derivatives section
END DO
!End loop over monomials
vtxgpureal(p,0,0,0) = SUM(p3r)
vtxgpuimag(p,0,0,0) = SUM(p3i)
END DO
!End loop over points

!$acc end region
PGI Compiler Messages:

161, Generating copyin(taylor_multidx(1:taylor_maxidx,0:ndir-1))
   Generating copyin(vtxgpureal(1:npoints,0,0,1:taylor_maxidx))
   Generating copyout(vtxgpureal(1:npoints,0,0,0:taylor_maxidx))
   Generating copyin(vtxgpuimag(1:npoints,0,0,1:taylor_maxidx))
   Generating copyout(vtxgpuimag(1:npoints,0,0,0:taylor_maxidx))
   Generating copyin(fimag(1:nterms))
   Generating copyin(freal(1:nterms))
   Generating copyin(route(1:order+1,1:npoints))
   Generating copyin(kimag(0:ndir-1,1:order+1,1:npoints))
   Generating copyin(yxv(0:ndir-1,1:order+1,1:nterms))
   Generating copyin(kreal(0:ndir-1,1:order+1,1:npoints))
   Generating copyin(taylor_maxidx(taylor_order))
   Generating compute capability 2.0 binary

165, Loop is parallelizable
   Accelerator kernel generated
165, !$acc do parallel, vector(256)
   CC 2.0 : 50 registers; 4 shared, 468 constant, 88 local memory bytes; 33
   occupancy

168, Loop carried dependence of 'diff_factor_imag' prevents parallelization
   Loop carried backward dependence of 'diff_factor_imag' prevents vectorization
   Loop carried dependence of 'pow_factor_real' prevents parallelization
   Loop carried backward dependence of 'pow_factor_real' prevents vectorization
   Loop carried dependence of 'pow_factor_imag' prevents parallelization
   Loop carried backward dependence of 'pow_factor_imag' prevents vectorization
169, sum reduction inlined
   Loop is parallelizable

170, sum reduction inlined
   Loop is parallelizable

171, Loop is parallelizable

172, sum reduction inlined
   Loop is parallelizable

181, Loop is parallelizable

185, Loop carried dependence of 'pow_factor_real' prevents parallelization
   Loop carried dependence of 'pow_factor_imag' prevents parallelization
Loop carried backward dependence of 'pow_factor_imag' prevents vectorization
   Inner sequential loop scheduled on accelerator
191, Loop is parallelizable
194, Loop carried scalar dependence for 'prod_pow_factor_real' at line 195
   Loop carried scalar dependence for 'prod_pow_factor_real' at line 198
   Scalar last value needed after loop for 'prod_pow_factor_real' at line 201
   Scalar last value needed after loop for 'prod_pow_factor_real' at line 202
   Loop carried scalar dependence for 'prod_pow_factor_imag' at line 198
   Loop carried scalar dependence for 'prod_pow_factor_imag' at line 199
   Scalar last value needed after loop for 'prod_pow_factor_imag' at line 201
   Scalar last value needed after loop for 'prod_pow_factor_imag' at line 202
   Inner sequential loop scheduled on accelerator
207, sum reduction inlined
   Loop is parallelizable
208, sum reduction inlined
   Loop is parallelizable
Appendix D

Modification to Work Plan

D.1 Analysis on a GPU Cluster

In the project preparation it was planned to build the application and test it on a GPU cluster. This version of the code will use MPI to communicate between the nodes. Access to Daresbury GPU cluster was also acquired for the same.

This plan was changed as this analysis was not performed. Emphasis was given to evaluating the directive based programming model and performance tuning on the new Fermi hardware. This is approved by the supervisors.

D.2 Porting application to OpenCL

As a part of the project preparation, one week of time was allocated for porting the application to OpenCL and analysing the performance. This activity was optional and was planned to be taken up only if time was available.

This was not taken up as there was not sufficient time to complete this. There were many unforeseen issues in evaluating the directive based approach due to bugs and issues in the PGI compiler used (10.6).
Appendix E

E.1 Timing data for application for nterms = 8000

<table>
<thead>
<tr>
<th>npoints</th>
<th>CUDAC-W1</th>
<th>CUDAC-W2</th>
<th>CUDAC-W3</th>
<th>CUDAC-W4</th>
<th>CUDAC-W5</th>
<th>CUDAC-W6</th>
<th>CUDAC-W7</th>
<th>CPU/PG F90</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>19.78</td>
<td>19.18</td>
<td>4.33</td>
<td>5.03</td>
<td>3.27</td>
<td>3.23</td>
<td>2.55</td>
<td>21.52</td>
</tr>
<tr>
<td>1024</td>
<td>20.68</td>
<td>20.12</td>
<td>9.55</td>
<td>8.44</td>
<td>5.45</td>
<td>5.42</td>
<td>5.13</td>
<td>43</td>
</tr>
<tr>
<td>2048</td>
<td>24.16</td>
<td>23.26</td>
<td>17.94</td>
<td>15.22</td>
<td>9.77</td>
<td>9.77</td>
<td>9.97</td>
<td>86.03</td>
</tr>
<tr>
<td>4096</td>
<td>35.66</td>
<td>34.69</td>
<td>35.09</td>
<td>28.99</td>
<td>18.61</td>
<td>18.59</td>
<td>20.13</td>
<td>172.64</td>
</tr>
<tr>
<td>8192</td>
<td>70.62</td>
<td>68.98</td>
<td>70.04</td>
<td>58.54</td>
<td>37.49</td>
<td>37.41</td>
<td>39.99</td>
<td>344.33</td>
</tr>
<tr>
<td>16384</td>
<td>140.8</td>
<td>137.19</td>
<td>141.05</td>
<td>116.41</td>
<td>74.66</td>
<td>74.48</td>
<td>80.32</td>
<td>691.62</td>
</tr>
</tbody>
</table>

Table 5 GPU kernel execution time in seconds for CUDAC code version on Ness; increasing npoints; nterms=8000 (includes time to allocate and copy memory, excludes time to initialize device)

<table>
<thead>
<tr>
<th>npoints</th>
<th>PGI-W1</th>
<th>PGI-W2</th>
<th>PGI-W3</th>
<th>CPU/P GF90</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>1.75</td>
<td>1.61</td>
<td>2.5</td>
<td>12.99</td>
</tr>
<tr>
<td>1024</td>
<td>1.93</td>
<td>1.82</td>
<td>2.58</td>
<td>26.6</td>
</tr>
<tr>
<td>2048</td>
<td>3.49</td>
<td>3.06</td>
<td>2.71</td>
<td>53.38</td>
</tr>
<tr>
<td>4096</td>
<td>13.03</td>
<td>10.64</td>
<td>9.34</td>
<td>103.92</td>
</tr>
<tr>
<td>8192</td>
<td>25.93</td>
<td>20.46</td>
<td>19.65</td>
<td>207.76</td>
</tr>
<tr>
<td>16384</td>
<td>52.96</td>
<td>42.13</td>
<td>39.15</td>
<td>416.97</td>
</tr>
</tbody>
</table>

Table 6 GPU kernel execution time in seconds for PGI code version on Fermi; increasing npoints; nterms=8000 (includes time to allocate and copy memory, excludes time to initialize device)
<table>
<thead>
<tr>
<th>Fermi</th>
<th>npoints</th>
<th>CUDAC/Initial</th>
<th>PGI/Accelerator</th>
<th>PGI/CPU</th>
<th>CUDAC/2-D</th>
<th>CUDAC/Block D</th>
<th>CPU/PGF90</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>512</td>
<td>7.28</td>
<td>2.5</td>
<td>2.46</td>
<td>2.48</td>
<td>1.33</td>
<td>12.99</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>7.36</td>
<td>2.58</td>
<td>4.93</td>
<td>3.94</td>
<td>2.66</td>
<td>26.6</td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td>7.5</td>
<td>2.71</td>
<td>9.84</td>
<td>7.56</td>
<td>5.26</td>
<td>53.38</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>9.47</td>
<td>9.34</td>
<td>19.67</td>
<td>14.1</td>
<td>10.48</td>
<td>103.92</td>
</tr>
<tr>
<td></td>
<td>8192</td>
<td>13.26</td>
<td>19.65</td>
<td>39.42</td>
<td>27.18</td>
<td>20.94</td>
<td>207.76</td>
</tr>
<tr>
<td></td>
<td>12288</td>
<td>26.53</td>
<td>28.97</td>
<td>54.21</td>
<td>40.27</td>
<td>31.37</td>
<td>320.79</td>
</tr>
<tr>
<td></td>
<td>16384</td>
<td>40.97</td>
<td>39.15</td>
<td>72.28</td>
<td>54.11</td>
<td>41.82</td>
<td>416.97</td>
</tr>
<tr>
<td></td>
<td>20480</td>
<td>52.04</td>
<td>49.26</td>
<td>97.16</td>
<td>67.18</td>
<td>52.27</td>
<td>519.2</td>
</tr>
<tr>
<td></td>
<td>24576</td>
<td>60.54</td>
<td></td>
<td></td>
<td>80.26</td>
<td>62.73</td>
<td>624.19</td>
</tr>
<tr>
<td></td>
<td>28672</td>
<td>71.43</td>
<td></td>
<td></td>
<td>93.34</td>
<td>73.18</td>
<td>726.87</td>
</tr>
<tr>
<td></td>
<td>32768</td>
<td>83.66</td>
<td></td>
<td></td>
<td>107.13</td>
<td>83.62</td>
<td>833.75</td>
</tr>
</tbody>
</table>

Table 7 GPU kernel execution time in seconds for code versions on Fermi; increasing npoints; nterms=8000 (includes time to allocate and copy memory, excludes time to initialize device). Initial, 2-D and Block D refers to the decompositions used.

<table>
<thead>
<tr>
<th>Ness</th>
<th>npoints</th>
<th>CUDAC/Initial</th>
<th>CUDAC/2-D</th>
<th>CUDAC/Block D</th>
<th>CPU/PGF90</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>512</td>
<td>19.18</td>
<td>3.23</td>
<td>2.55</td>
<td>21.52</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>20.12</td>
<td>5.42</td>
<td>5.13</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td>23.26</td>
<td>9.77</td>
<td>9.97</td>
<td>86.03</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>34.69</td>
<td>18.59</td>
<td>20.13</td>
<td>172.64</td>
</tr>
<tr>
<td></td>
<td>8192</td>
<td>68.98</td>
<td>37.41</td>
<td>39.99</td>
<td>344.33</td>
</tr>
<tr>
<td></td>
<td>12288</td>
<td>103.61</td>
<td>55.27</td>
<td>60.13</td>
<td>516.49</td>
</tr>
<tr>
<td></td>
<td>16384</td>
<td>137.19</td>
<td>74.48</td>
<td>80.32</td>
<td>691.62</td>
</tr>
<tr>
<td></td>
<td>20480</td>
<td>169.6</td>
<td>92.09</td>
<td>99.46</td>
<td>858.66</td>
</tr>
<tr>
<td></td>
<td>24576</td>
<td>202.98</td>
<td>111.61</td>
<td>119.07</td>
<td>1023.46</td>
</tr>
<tr>
<td></td>
<td>28672</td>
<td>237.64</td>
<td>128.61</td>
<td>140.9</td>
<td>1205.17</td>
</tr>
<tr>
<td></td>
<td>32768</td>
<td>277.43</td>
<td>147.81</td>
<td>161.15</td>
<td>1377.2</td>
</tr>
</tbody>
</table>

Table 8 GPU kernel execution time in seconds for code version on Ness; increasing npoints; nterms=8000 (includes time to allocate and copy memory, excludes time to initialize device) Initial, 2-D and Block D refers to the decompositions used.
### E.2 Timing data for application with spin and color changes

(nteams= 8000)

<table>
<thead>
<tr>
<th>npoints</th>
<th>CUDAC-SC1</th>
<th>CUDAC-SC2</th>
<th>CUDAC-SC3</th>
<th>CPU/PGF90</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>9.08</td>
<td>29.97</td>
<td>7.45</td>
<td>13.22</td>
</tr>
<tr>
<td>1024</td>
<td>9.21</td>
<td>59.82</td>
<td>7.63</td>
<td>26.12</td>
</tr>
<tr>
<td>2048</td>
<td>9.71</td>
<td>119.46</td>
<td>7.95</td>
<td>52.23</td>
</tr>
<tr>
<td>4096</td>
<td>14.25</td>
<td>238.4</td>
<td>10.39</td>
<td>108.15</td>
</tr>
</tbody>
</table>

Table 9 GPU kernel execution time in seconds for code version with spin and color changes on Fermi; increasing npoints; nterms=8000 (includes time to allocate and copy memory, excludes time to initialize device)

<table>
<thead>
<tr>
<th>npoints</th>
<th>CUDAC-SC1</th>
<th>CUDAC-SC3</th>
<th>CPU/PGF90</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>21.12</td>
<td>19.4</td>
<td>22.19</td>
</tr>
<tr>
<td>1024</td>
<td>22.9</td>
<td>20.42</td>
<td>43.29</td>
</tr>
<tr>
<td>2048</td>
<td>29.43</td>
<td>24.08</td>
<td>86.61</td>
</tr>
<tr>
<td>4096</td>
<td>52.56</td>
<td>36.3</td>
<td>173.19</td>
</tr>
</tbody>
</table>

Table 10 GPU kernel execution time in seconds for code version with spin and color changes on Fermi; increasing npoints; nterms=8000 (includes time to allocate and copy memory, excludes time to initialize device)

### E.3 Timing data for application for npoints = 4096

<table>
<thead>
<tr>
<th>Nterms</th>
<th>CUDAC/Initial</th>
<th>CUDAC/2-D</th>
<th>CUDAC/Block D</th>
<th>PGI/Accelerator</th>
<th>PGI/CPU</th>
<th>CPU/PGF90</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.12</td>
<td>0.36</td>
<td>1.79</td>
<td>0.17</td>
<td>0.25</td>
<td>1.34</td>
</tr>
<tr>
<td>200</td>
<td>0.24</td>
<td>0.53</td>
<td>1.9</td>
<td>0.29</td>
<td>0.5</td>
<td>2.6</td>
</tr>
<tr>
<td>400</td>
<td>0.47</td>
<td>0.86</td>
<td>2.12</td>
<td>0.98</td>
<td>0.98</td>
<td>5.2</td>
</tr>
<tr>
<td>800</td>
<td>0.95</td>
<td>1.55</td>
<td>2.58</td>
<td>1.9</td>
<td>3.89</td>
<td>20.77</td>
</tr>
<tr>
<td>1600</td>
<td>1.89</td>
<td>2.95</td>
<td>3.42</td>
<td>7.5</td>
<td>15.52</td>
<td>83.31</td>
</tr>
<tr>
<td>3200</td>
<td>3.8</td>
<td>5.74</td>
<td>5.15</td>
<td>3.77</td>
<td>7.76</td>
<td>41.53</td>
</tr>
<tr>
<td>6400</td>
<td>7.58</td>
<td>11.32</td>
<td>8.68</td>
<td>14.95</td>
<td>31.02</td>
<td>166.16</td>
</tr>
<tr>
<td>12800</td>
<td>15.15</td>
<td>22.45</td>
<td>15.77</td>
<td>29.93</td>
<td>62.18</td>
<td>341.45</td>
</tr>
<tr>
<td>25600</td>
<td>30.31</td>
<td>44.77</td>
<td>29.77</td>
<td>58.25</td>
<td>ERR</td>
<td>ERR</td>
</tr>
<tr>
<td>51200</td>
<td>60.61</td>
<td>89.38</td>
<td>ERR</td>
<td>ERR</td>
<td>ERR</td>
<td>ERR</td>
</tr>
</tbody>
</table>

Table 11 GPU kernel execution time in seconds for code version on Fermi; increasing nterms; npoints=4096; (includes time to allocate and copy memory, excludes time to initialize device)
<table>
<thead>
<tr>
<th>nterms</th>
<th>CUDAC/Initial</th>
<th>CUDAC/2-D</th>
<th>CUDAC/Block D</th>
<th>CPU/PGF90</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.44</td>
<td>1.06</td>
<td>3.26</td>
<td>2.17</td>
</tr>
<tr>
<td>200</td>
<td>0.87</td>
<td>1.29</td>
<td>3.32</td>
<td>4.32</td>
</tr>
<tr>
<td>400</td>
<td>1.75</td>
<td>1.71</td>
<td>3.77</td>
<td>10.12</td>
</tr>
<tr>
<td>800</td>
<td>3.49</td>
<td>2.59</td>
<td>4.61</td>
<td>17.17</td>
</tr>
<tr>
<td>1600</td>
<td>6.95</td>
<td>4.38</td>
<td>6.31</td>
<td>34.45</td>
</tr>
<tr>
<td>3200</td>
<td>13.89</td>
<td>7.96</td>
<td>9.62</td>
<td>69.98</td>
</tr>
<tr>
<td>6400</td>
<td>27.78</td>
<td>15.09</td>
<td>16.23</td>
<td>138.93</td>
</tr>
<tr>
<td>12800</td>
<td>56.09</td>
<td>29.37</td>
<td>29.93</td>
<td>275.46</td>
</tr>
<tr>
<td>25600</td>
<td>112.19</td>
<td>57.97</td>
<td>57.61</td>
<td>561.05</td>
</tr>
<tr>
<td>51200</td>
<td>224.38</td>
<td>115.16</td>
<td>113.01</td>
<td>1102.198</td>
</tr>
</tbody>
</table>

Table 12 GPU kernel execution time in seconds for code version on Ness; increasing nterms; npoints=4096; (includes time to allocate and copy memory, excludes time to initialize device)
References


