An Acceleration of MUSE using GPU

Perri Needham

27/08/10
Abstract

GPGPU (General Purpose Graphical Processing Unit) technology has enabled the scientific community to harness the great computational power of the GPU (Graphical Processing Unit). In this dissertation, an investigation into accelerating an in-house program (MUSE) from Manchester University’s Computational Chemistry department was performed. The code was ported on the NVIDIA Fermi GPU and the NVIDIA Tesla C1060 GPU, to investigate the performance differences between the two architectures. The code was also ported using the C for CUDA API (Application Programming Language) and the CUDA Fortran API, to investigate the performance differences between the two APIs. After initially porting the code, no improvement to the performance was obtained, the execution time actually increased. The code was optimised in order to exploit the GPU’s architectural features, however, a performance improvement was still not obtained compared to running the code on a CPU. The performance of the different architectures, and also the different APIs, was benchmarked with a matrix multiplication test code. The results of the benchmarking showed that the NVIDIA Fermi out-performed the NVIDIA Tesla C1060 when data transfers were excluded, and the C for CUDA API out-performed the CUDA Fortran API.
Contents

1 Introduction .................................................................................................................. 1
2 Background .................................................................................................................. 2
  2.1 GPGPU technology ............................................................................................... 2
  2.2 Architectures ........................................................................................................ 4
  2.3 Compute Capability .............................................................................................. 4
  2.4 Application Programming Interfaces (APIs) ..................................................... 4
    2.4.1 C for CUDA ................................................................................................... 5
    2.4.2 CUDA Fortran ............................................................................................... 12
    2.4.3 CUBLAS Library ......................................................................................... 16
  2.5 Code suitability for acceleration using a GPU ...................................................... 16
  2.6 MUSE .................................................................................................................... 18
3 Benchmarking ............................................................................................................ 20
  3.1 Benchmarking the architectures ........................................................................... 20
  3.2 Benchmarking the APIs ....................................................................................... 26
4 Porting ........................................................................................................................ 28
  4.1 Identifying code and data for porting ................................................................. 28
  4.2 Converting to C ................................................................................................... 31
  4.3 Decomposing into threads ................................................................................... 34
  4.4 CUBLAS ............................................................................................................. 34
  4.5 Debugging ............................................................................................................. 35
5 Optimisation ............................................................................................................... 36
  5.1 C for CUDA optimisation on the C1060 ............................................................ 36
  5.2 CUDA FORTRAN optimisation on the Fermi ................................................... 43
6 Analysis ...................................................................................................................... 46
  6.1 Performance comparison of the GPU architectures ........................................... 46
  6.2 Performance comparison of the APIs ................................................................... 47
6.3 Performance comparison between the CPU and the GPU architectures  

7 Conclusion ......................................................................................................................... 50

7.1 Summary of findings ..................................................................................................... 50

7.2 Future work .................................................................................................................. 51

Appendix A: C for CUDA Benchmark .................................................................................... 52

Appendix B: CUDA FORTRAN Benchmark ............................................................................ 53

Appendix C: CUDA Arrays .................................................................................................. 54

Appendix D: Block size execution times .............................................................................. 55

8 References ........................................................................................................................ 57
List of Figures

Figure 2.1 A graph to compare the advances in peak clock frequency of CPUs and GPUs over time, courtesy of NVIDIA [1].........................................................2

Figure 2.2 A diagram showing the main components of a CUDA GPU architecture [1]..................................................................................................................3

Figure 2.3 An illustration of the thread hierarchy employed in CUDA architecture. 6

Figure 2.4 Example code of a vector-vector multiplication in C......................................................8

Figure 2.5 Example host code of a vector-vector multiplication.........................................................9

Figure 2.6 Example device code of a vector-vector multiplication kernel.........................11

Figure 2.7 Diagram to illustrate how a half-warp of threads accesses array elements in shared memory banks...................................................................................12

Figure 2.8 Example host code for a vector-vector multiplication using CUDA Fortran.......................................................................................................................15

Figure 2.9 Example device code for a vector-vector multiplication using CUDA Fortran.......................................................................................................................15

Figure 3.1 Fermi memory hierarchy vs. Tesla C1060 memory hierarchy..............22

Figure 3.2 Graph to show the fastest execution times of a matrix multiplication benchmark, out of 5 runs, on the C1060 and the Fermi architectures.............23

Figure 3.3 Diagram to illustrate the matrix multiplication code design...............24

Figure 3.4 Graph to show the fastest execution times of a matrix multiplication benchmark, out of 5 runs, using the C for CUDA and the CUDA Fortran APIs, on the Fermi architecture..............................................................26

Figure 4.1 Execution time of functions in MUSE which, used more than 1% of the total execution........................................................................................................28

Figure 4.2 Fortran 77 code, the computation part of the matmul function of MUSE. .......................................................................................................................................30

Figure 4.3 Diagram to show the order a 3x3 array is stored in memory, in Fortran and C.................................................................32

Figure 4.4 Pseudo Fortran calling code...............................................................33

Figure 4.5 Pseudo C for CUDA host function code.............................................33

Figure 4.6 C code replacement for $ij$ counter......................................................34
**Figure 5.1** Graph to show the execution times of the *matmul* function before and after porting to GPU (excluding data transfers). .................................................................36

**Figure 5.2** Graph to show the execution times of the *matmul* function before and after porting to GPU (including data transfers). .................................................................37

**Figure 5.3** Graph to show the execution times of the *matmul* function: before porting; after the initial port; after the use of texture memory (arrays a and b); and after the use of shared memory (array d) excluding data transfers. .........................39

**Figure 5.4** Graph to show the execution times of the *matmul* function: before porting; after the use of texture memory (arrays a and b); and after reducing data transfers, all including data transfers. .................................................................41

**Figure 5.5** Graph to show the execution times of the *matmul* function: before porting-original; after the use of shared memory (array ifact), 16 threads per block-shared ifact 16; after the use of texture memory (arrays a and b)-texture; after the use of shared memory (array ifact), 3 threads per block-shared ifact 3, all excluding data transfers. .................................................................42

**Figure 5.6** Graph to show the execution times of the *matmul* function: before porting-original; after porting-initial port; after merging the two *matmul* calls-merged calls; after putting the *ifact* array into constant memory-constant memory; and after putting arrays *a* and *b* into shared memory-shared memory, all including data transfers. .................................................................43

**Figure 5.7** Graph to show the execution times of the *matmul* function: before porting-original; after porting-initial port; after merging the two *matmul* calls-merged calls; after putting the *ifact* array into constant memory-constant memory; and after putting arrays *a* and *b* into shared memory-shared memory, all excluding data transfers. .................................................................44

**Figure 6.1** Graph to show the average execution times of the *matmul* function: on the CPU- original; on the Fermi GPU in C for CUDA with data transfers-Fermi including xfers, and without data transfers- Fermi excluding data xfers; on the C1060 GPU in C for CUDA with data transfers- C1060 including xfers, and without data transfers- C1060 excluding data xfers. .................................................................46

**Figure 6.2** Graph to show the average execution times of the *matmul* function: on the CPU- original; on the Fermi GPU in C for CUDA with data transfers- C for CUDA including xfers, and without data transfers- C for CUDA excluding data xfers; on the Fermi GPU in CUDA Fortran with data transfers- CUDA Fortran including xfers, and without data transfers- CUDA Fortran excluding data xfers. .................................................................48

**Figure D.1** Graph to show the average execution times of the *matmul* function excluding data transfers, with varying numbers of threads per block. .................................................................55

**Figure D.2** Graph to show the average execution times of the *matmul* function excluding data transfers for block sizes 1-4. .................................................................56
List of Tables

Table 5.1 Results of the experiment to test the effect of the overheads when using built-in vector types. ........................................................................................................................................... 40
Acknowledgements

I’d like to thank Xu Guo and Fiona Reid for their advice and supervision throughout the duration of this MSc project. I’d also like to thank Dr Burton for his support, along with all the EPCC staff for their guidance and patience.
1 Introduction

Dr Burton’s Computational Chemistry research group at Manchester University has developed a Fortran 77 code to compute the electronic configuration of molecular systems, namely MUSE. GPUs (Graphical Processing Units) are primarily designed to process computer graphics; however, the scientific community has recently exploited its computational power in HPC applications. An investigation into the use of GPUs as a method of improving the performance of MUSE was to be carried out.

The primary goal in this dissertation was to improve the performance of MUSE by porting it to GPU. Development was to be carried out on the only available GPU, Tesla C1060, which is augmented on the HPC facility Ness. This GPU does not support the API (Application Programming Interface) required to port the code in its native language (CUDA Fortran), hence, code sections had to be converted to the C programming language in order to be ported to the GPU. Mid-way through the project a new GPU became available on the system. This did support the API required to port the code in its native language, and so the project took an interesting new direction. MUSE was to be ported using both APIs on both architectures and the performance implications compared. In order for a fair comparison between the two architectures and the two APIs, benchmarking was carried out in addition to porting MUSE. The results of which could be used as a guideline to the expected performance difference when analysing the ported MUSE application.

This dissertation gives a fully detailed description of GPU technology, the different architectures being compared, the APIs being compared, the suitability of applications for use with a GPU and the application being ported, MUSE, in chapter 2. Chapter 3 gives details and explanations of the benchmarking carried out, along with the differences between the two architectures. In chapter 4, the process of porting MUSE is discussed in step-by-step form. After porting the code, optimisations are carried out. These are described and analysed in chapter 5. An overall analysis of the results from porting on the different architectures, using the different APIs is carried out in chapter 6, along with a discussion of the performance differences of MUSE when executed on the CPU and the GPU. Finally, the dissertation is concluded in chapter 7.
2 Background

This chapter gives some background information on the key components of the project, namely, GPGPU technology, Architectures, Application Programming Interfaces (APIs), code suitability for acceleration using GPUs and the code to be accelerated (MUSE).

2.1 GPGPU technology

The clock frequency of a core on a CPU can no longer be increased due to physical limitations such as heat dissipation. In order for CPU performance to adhere to Moore’s Law, an alternative has been to simply increase the number of cores per chip. But, with this advance comes increased complexity for the programmer as it becomes more and more difficult for applications to exploit this technology.

Graphical Processing Units (GPUs) are specifically designed for the computer gaming industry. The graphics on computer games require repeatedly calculating the pixel value and little else, so in place of sophisticated cache systems, controllers and other complex features, GPUs have hundreds of simplistic cores. GPUs are fundamentally parallel with each core computing as an individual thread, ideal for data decompositions with few dependencies between threads. As a result, GPUs can provide a great amount of high performance, parallel computing power. As can be seen from figure 2.1, they provide a means to be able to improve performance of suitable code far beyond the limits of a CPU. This processing power has recently been harnessed by the scientific community in the form of General Purpose computation on Graphical Processing Units (GPGPU).

![Figure 2.1](image)

**Figure 2.1** A graph to compare the advances in peak clock frequency of CPUs and GPUs over time, courtesy of NVIDIA [1]
GPGPUs have been specifically designed to target HPC applications as they have some additional features to the GPUs used in the computer gaming industry such as, caching technology and ECC (error correcting code) support. The GPU is used as an accelerator, in conjunction with the CPU, working on specifically selected functions of the code (known as kernels). The majority of the code still runs on the CPU, as most scientific code is not suitable to be run on GPUs. The factors that determine whether a code is suitable to be ported to a GPU are discussed further in section 2.5. The use of the GPGPU as an accelerator has lead to massive performance improvements for particular codes [2].

In the CUDA (Compute Unified Device Architecture) programming model (see section 2.4.1) the CPU is described as the host and the coupled GPU, the device. The host and the device are connected via a PCIe (Peripheral Component Interconnect express) cable and data is passed between the host’s memory and the device’s memory. The GPU is partitioned into streaming multiprocessor (SM) units. Each SM contains eight scalar processor (SP) cores, one double-precision unit, two special function units and a multithreaded instruction unit for scheduling the warps. There are four types of on-chip memory on each multiprocessor: each SP has a set of 32-bit registers; all SP cores have a shared memory space (a parallel data cache); all SP cores share a constant cache which is read-only and enables the speed of reads from constant memory space (a read-only component of device memory) to be increased; all SP cores share a texture cache which is read-only and enables the speed of reads from texture memory space (a read-only component of device memory) to be increased. Figure 2.2 shows a diagram of how the memory layout can be visualised by the developer [1].

![Diagram of CUDA GPU architecture](image)

**Figure 2.2** A diagram showing the main components of a CUDA GPU architecture [1].
2.2 Architectures

The two main vendors which, dominate the GPU market, are currently NVIDIA and AMD. Intel was due to break into the GPU market this year with their GPU chip, Intel Larrabee, but the release has been delayed.

The GPUs used to carry out this dissertation were both NVIDIA products. The HPC facility Ness is augmented with the GPU nodes NVIDIA 10-series Tesla C1060 and the latest NVIDIA Tesla, 20-series C2050 (coda name Fermi).

The Tesla C1060 has 240 SIMD cores. It’s peak performance for single precision is 933 GFLOPS, whereas, it’s peak performance for double precision is 78 GFLOPS. The total dedicated memory it has available is 4 GDDR3 [3]. With the release of the new Fermi card, NVIDIA have almost doubled the number of SIMD cores to 448. The peak single precision performance has been increased to 1.03 TFLOPS and the peak double precision performance has been increased to 515 GFLOPS. The total available memory is 3GB GDDR5 [4]. The primary API for all NVIDIA GPU’s is the CUDA APIs, C for CUDA and PGI’s CUDA Fortran. Other API’s are also supported such as OpenCL (Open Compute Language), Rapidmind and DirectCompute.

The latest technology from AMD is the AMD Firestream 9370, which is due to be released in the 3rd quarter of 2010. It boasts 1,600 SIMD (single instruction, multiple data) cores, a single precision peak performance of 2.64 TFLOPS, a double precision peak performance of 528 GFLOPS, and 4GB of high-speed GDDR5 (Graphic Double Data Rate, version 5) memory, providing an enormous amount of computing power, making this the leading GPU in the market for performance. OpenCL, DirectX11 and OpenGL are some of the APIs, which will be supported by the new Firestream [5].

2.3 Compute Capability

The compute capability of a device describes the core architecture of a device by its major revision number, and the improvements to the core architecture by its minor revision number. The Tesla C1060 has a compute capability of 1.3. The major revision number is 1.x, and the minor revision number is x.3. The Fermi card has a compute capability of 2.0. There are some distinct differences between the two architectures. These will be discussed further in chapter 3 [1].

2.4 Application Programming Interfaces (APIs)

There are many different APIs available for programming GPUs, each exploiting different styles and levels of programming. The propriety interface to the NVIDIA CUDA architectures is C for CUDA. PGI provide Fortran support with the PGI CUDA Fortran API, which enables Fortran programmers to program CUDA directly. These two APIs were used in this research and will be discussed further in this chapter.
The main alternative to the CUDA APIs is OpenCL (Open Compute Language). OpenCL is supported by both NVIDIA and AMD. It offers a lower level of programming to developers compared with CUDA APIs, giving programmers more control i.e. manual memory management. Different terminology is used to describe the programming model, for example, what is known as a thread in the CUDA APIs is termed a work item in OpenCL. However, the basic functionality is very similar to that of the CUDA APIs [5].

Other alternatives include the high-level development platform Rapidmind, which is a set of C++ libraries, and Microsoft’s windows-only DirectCompute API.

### 2.4.1 C for CUDA

The details in this chapter have been referenced from the NVIDIA resource the CUDA Programming Guide [1].

CUDA is NVIDIA’s general-purpose parallel computing architecture. C for CUDA is one of the APIs being used in this dissertation to port MUSE to GPU. C for CUDA is a high-level programming language and with it comes a new parallel programming model.

The C for CUDA programming interface is designed to be accessible to users of all abilities and so is made up of a small number of extensions to the C programming language and a runtime library made up of three components; host, device and common. The host component contains functions to access and control the device, the device component contains functions specific to the device and the common component contains built-in vector types and functions supported by both the device and the host, such as fetching texture memory.

The extensions to the C programming language fall into four categories: function type qualifiers; variable type qualifiers; kernel directive; and the built-in thread and block indexing variables.

**Function type qualifiers**

- **__device__** Specifies that a function is called and executed on the device.
- **__global__** Specifies that a function is the kernel to be called from the host and executed by the device.
- **__host__** Specifies that a function is called and executed on the host.

**Variable type qualifiers**

- **__device__** Specifies that a variable resides on the device. When no other variable type qualifiers are used with it, i.e. shared, the variable resides in global memory.
- **__constant__** Specifies that a variable resides in constant memory on the device.
- **__shared__** Specifies that a variable resides in shared memory on the device.
Kernel directive

In the CUDA programming model, functions within the C code are offloaded to the GPU as kernels, where they are executed in parallel N times by N threads. To declare a kernel the declaration specifier, `__global__`, is used. To specify the number of CUDA threads for each call the syntax, `<<<dimGrid, dimBlock>>`, is used. The threads are organised into blocks of threads within a grid of blocks. Figure 2.3 demonstrated how the threads are organised within the thread hierarchy.

![Figure 2.3](image-url) An illustration of the thread hierarchy employed in CUDA architecture.

Built-in variables

The grid dimensions are specified by the first parameter (dimGrid) and the number of threads per block is specified by the second parameter (dimBlock). Built-in variables allow ease of navigation through the threads in the grid and/or blocks. The `threadIdx` variable is a vector with an extent of 3, which allows navigation through a one- two- or three-dimensional thread block. The `blockIdx` variable is a vector with an extent of 2, which identifies the position of the block in the grid of blocks using a one- or two- dimensional index. The dimension of the block can be accessed using the blockDim variable, which again is a 2-component vector.
There are various functions available to the user, which can be called from either host code or device code. Some of the most crucial functions needed in order to port code to the GPU are those, which manage memory. Functions to allocate, transfer and deallocate memory are described below.

**Runtime functions**

**cudaMalloc()** Allocates linear memory on the device. It is recommended that for 2D and 3D arrays, memory should be allocated using `cudaMallocPitch()` and `cudaMalloc3D`, respectively. This is because these functions pad the array in memory, ensuring they are aligned such that the best performance can be seen when accessing arrays.

**cudaMemcpy()** Transfers linear memory between the host and the device. The direction of the transfer is passed to the function as an argument. **cudaMemcpy2DToArray()** allows 2D arrays to be transferred to CUDA arrays. CUDA arrays on the device are read-only memory layouts, optimised for texture fetching. They can be written but, from host code only. A full explanation of CUDA arrays can be found in Appendix D.

**cudaFree()** Deallocates memory on the device.

**Memory**

There are three separate memory spaces that CUDA threads have access to: global memory; shared memory; local memory. The programmer has control over managing global and shared memory, however data is allocated to local memory by the compiler. Local memory is private to each thread and lasts the lifetime of the thread; shared memory is private to each block of threads and has the same lifetime as the block; global memory can be accessed by all threads and has the lifetime of the application unless it is deallocated by the host code. There are also two read-only memory spaces, which have a lifetime of the application: constant memory; texture memory. Memory is managed through calls to the CUDA runtime library. Device memory is allocated and deallocated by the CPU. Memory transfers between the host and the device along with texture and constant memory management, are again, performed by the host. Shared memory is managed in device code using the variable type qualifier, `__shared__`.

Threads within the same threadblock are expected to be located on the same processing core. This enables communication between threads in the same block by the use of a shared memory (which is the equivalent of an L1 cache) and the intrinsic function `__syncthreads()`. This function allows threads to synchronise their execution by acting as a barrier, preventing further execution until all threads have reached the barrier. This enables data in the shared memory to be coordinated before continuing the execution of the kernel. The requirement that all threads in a block have to be on the same core, however, does create a restriction on the number of threads in a block. This is due to the limitations of the shared memory. Currently, a maximum of 512 threads per block are allowed due to the resources available to each core.
To demonstrate how the C for CUDA extensions and runtime functions are used, a simple vector-vector multiplication problem will be used as an example.

**Vector-vector multiplication example**

This example shows; how a section of code can be modified into a kernel for execution on a GPU; how data is transferred between the host and the device; how memory can be managed on the device to optimise performance. In this project, a vector-vector multiplication function was ported to use the GPU. This example will run through a more simplistic version of the vector-vector multiplication. *Figure 2.4* shows the original CPU code for the vector multiplication. The function loops over the array index, multiplying each element in array A by the equivalent element in array B, assigning the result to the equivalent element in array C.

```c
#define M=1024

void vector_multiplication(float *A, float *B, float *C) {
    for(int i=0; i<M; i++)
    {
        C[i]=A[i]*B[i];
    }
    return;
}
```

*Figure 2.4* Example code of a vector-vector multiplication in C.

To execute this kernel on a GPU, the variables required to be transferred to the device from the host need to be identified. Next the variables need to be allocated on the device and the data required for the calculation, transferred to the device. Once the data has been successfully transferred, the problem needs to be decomposed into many-threads. Any inter-thread dependencies need to be identified and either, the code needs to be modified to remove the dependencies or the GPU code needs to be developed in a way so as to account for the dependencies, i.e. through the use of the `__sync_thread()` runtime function. Having decomposed the data into threads the next step is to invoke the kernel, which executes on the device. In the device function, the memory location of the data needs to be managed to ensure the best performance. Once the kernel has executed on the device the host code needs to transfer the results back to the host and deallocate the memory on the device.

*Figure 2.5* shows how the host code was implemented for this example. The host code is run on the CPU. The arrays A, B and C are declared with an extent of 1024. This size was purposely chosen as it divides by the blocksize (32) exactly. The blocksize variable holds the value of the number of threads in each block.
The first function call in the host code refers to an external function (which is not shown here) responsible for populating arrays A and B with random numbers. Pointers to device arrays Ad, Bd and Cd are declared. These arrays are then allocated on the device using the runtime function `cudaMalloc`. This is a synchronous function, which means the threads all wait until every thread has carried out the operation, before continuing execution of the code. Once allocated, the data in arrays Ad and Bd is transferred to the device using the synchronous `cudaMemcpy` function.

```c
#define M=1024
#define blocksize=32

void main()
{
  int blocksize; M
  float A[M], B[M], C[M];
  random(A, B, M);
  float *Ad, *Bd, *Cd;
  cudaMemcpy((void**)&Ad, M);
  cudaMemcpy((void**)&Bd, M);
  cudaMemcpy((void**)&Cd, M);
  cudaMemcpy(Ad, A, M, cudaMemcpyHostToDevice);
  cudaMemcpy(Bd, B, M, cudaMemcpyHostToDevice);

  dim3 dimBlock(blocksize);
  int N;
  if (((M/blocksize)==0){
    N=M/blocksize;
  }
  else{
    N-M/blocksize+1;
  }
  dim3 dimGrid(N);
  vector_multiplication<<<dimGrid, dimBlock>>>(Ad, Bd, Cd, blocksize, M);
  cudaThreadSynchronize();
  cudaMemcpy(C, Cd, M, cudaMemcpyDeviceToHost);
  cudaFree(Cd);
  cudaFree(Ad);
  cudaFree(Bd);
}
```

**Figure 2.5** Example host code of a vector-vector multiplication.

Choosing the execution parameters `dimblock` and `dimGrid` requires consideration of the hardware. In order to attain maximum performance, a good balance between hiding the latency and exploiting the resources needs to be found. This is mainly
done by experimentation with the execution parameters. However, some guidelines can be followed to help with making the choice. The number of blocks in a grid should be large enough so as each multiprocessor has at least one block to execute. The GPU with the maximum number of multiprocessors that this project utilises, has 30 multiprocessors. Choosing a blocksize of 32 means that there are 32 blocks, each with exactly 32 active threads. This adheres to each multiprocessor having at least one block to execute. As threads are scheduled in warps (groups of 32 threads); maximum efficiency is achieved when all threads in a warp are executing the same path as each other. Selecting a blocksize of 32 threads should minimise the amount of time threads spend idle and aid coalescing. Coalescing is when loads/stores to/from memory locations for each thread are grouped together, resulting in one large memory transfer, as opposed to several small transfers. The aim is to achieve around 50% occupancy on the multiprocessor. However, as multiprocessors can execute multiple blocks simultaneously, the potential occupancy of a multiprocessor needs to be considered further. If the occupancy is greater than 50% the general trend is that, due to less registers being available per core, the local memory is used instead. This can hinder performance improvement.

The execution parameters are then used to invoke the kernel launch using the special CUDA syntax. The variables required by the kernel are passed in as arguments. It’s important to note here that pointers are not allowed unless they point to device memory, as the device does not have access to the host’s memory.

As the kernel launch is asynchronous i.e. control is immediately passed back to the CPU after the kernel is launched, before the results can be copied back into the C array, the threads need to be synchronised with the use of the cudaThreadSynchronize() runtime function. Once the results array has been copied back into the C array, the device arrays can be deallocated using the cudaFree() function.

*Figure 2.6* shows the kernel code, which executes on the device. To declare a device function the __global__ function qualifier is required. Each thread in the block of threads is assigned an index value using the built-in threadIdx.x and blockIdx.x variables. Each thread is assigned two variables: its global thread identification, i; its local thread identification within the block, tx.

As uncached shared memory latency can be 100x lower than global memory latency, to get the best performance, shared memory space should be utilised as efficiently as possible. This however, is reliant on their being no bank conflicts. Banks are equally sized sections of shared memory, which are accessed simultaneously by the threads in a half warp. Bank conflicts are where more than one memory address, that is requested, points to the same bank. In this eventuality, the memory accesses are serialised, having significant affects on performance. This will be discussed further, later on.

In this example, two shared arrays are created (Asub and Bsub) in which, the sub-section of the matrix (that a particular block is operating on) is assigned. As the number of threads in a block is 32, the number of array elements in the sub-sections
is also 32. The corresponding sections of the global matrices are loaded into $A_{sub}$ and $B_{sub}$ with each thread loading an individual element. This ensures that the

```c
__global__ void vector_multiplication(float *A, float *B, float *C, blocksize, N) {
    int i = blockIdx.x * blocksize + threadIdx.x;
    int tx = threadIdx.x;

    __shared__ float A_sub[blocksize];
    __shared__ float B_sub[blocksize];

    A_sub[tx] = A[i];
    B_sub[tx] = B[i];

    float C_sub;

    if (i < N) {
        C_sub = A_sub[tx] * B_sub[tx];
    }

    C[i] = C_sub;
}
```

**Figure 2.6** Example device code of a vector-vector multiplication kernel.

global memory reads are coalesced. Coalescing is where all threads access data in memory simultaneously, increasing the bandwidth of a transaction. Global memory read/writes can be coalesced into one transaction for a half-warp of threads. To ensure coalescing on a GPU with a compute capability of 1.2 or higher, when using single precision, the words being accessed by the threads have to lie in the same 128-byte section. As single precision floating-point data occupies 4 bytes of memory, 32 elements will lie in a 128-byte section, hence in this example, the global reads/writes should be coalesced.

A conditional if statement is inserted around the calculation. This ensures that threads with an identification number higher than the extent of the array, do not carry out the calculation and over-write parts of memory. Each thread carries out the calculation on one array element and assigns the result to a scalar variable. Each thread then loads the scalar variable into the corresponding position in the C array in global memory. As discussed before, reads to global memory should be coalesced.

When each thread requests an element from the $A_{sub}/B_{sub}$ array in shared memory, it is important to avoid bank conflicts as discussed earlier. To do this, an understanding of how data is mapped to the memory banks is required. The memory banks are organised so that consecutive 32-bit words are stored in consecutive banks. **Figure 2.7** shows how the array elements in this example are organised into banks. 2 half-warps of threads access 32 x 32-bit words, organised into 16 banks.
Figure 2.7 Diagram to illustrate how a half-warp of threads accesses array elements in shared memory banks.

This arrangement results in no bank conflicts when fetching single precision elements. This is because a request for shared memory is split into a request for each half-warp of threads, and there can be no bank conflict between separate half-warps.

Further optimisations

In the above example, the code has been optimised to coalesce global memory loads/stores, make use of the high bandwidth shared memory space, eliminate any bank conflicts when accessing shared memory, and find a good balance between occupancy and latency. These are only a few of the optimisations available to the developer. The optimisations carried out as part of this project shall be discussed, in detail, in chapter 5.

2.4.2 CUDA Fortran

A large proportion of scientific programmers choose to develop their high performance code using the Fortran programming language, due to its convenient array functionality, amongst other features. PGI and NVIDIA teamed up to develop the PGI CUDA Fortran compiler, enabling Fortran-users to harness the performance benefits of the GPU. The programming model for CUDA Fortran is very similar to that of the C for CUDA model, however there are some subtle differences. This chapter will discuss the main differences between CUDA Fortran and C for CUDA. To demonstrate how to implement a CUDA Fortran code, a simple vector-vector multiplication problem will be described as an example. The PGI CUDA Fortran programming guide was used as a reference for this chapter [7].

Function and Variable type qualifiers

To define any function or variable in CUDA Fortran, the Fortran 90 attributes() feature is used. The attributes available for functions are global, host and device, as in C for CUDA. Again, as in C for CUDA, variables can be attributed with the qualifiers device, constant and shared. Additionally, CUDA Fortran has appended the pinned variable qualifier. This allows allocatable arrays to be allocated to page-locked memory on the host, availability pending. The benefit of this is that memory
transfers between the host and the device can be carried out faster when memory is page-locked as opposed to paged. An unspecified variable in host code will be stored in host memory by default.

When passing arguments to a subroutine/function, it is Fortran convention to pass the pointers to the variables as arguments, and not to pass the actual value of the variable. As the device cannot access host memory, values of variables need to be explicitly transferred. An additional attribute has been included to deal with this. Scalar variables can be declared with the value attribute. This tells the compiler to pass the actual value of the argument to the function, and removes any need to explicitly copy scalar variable from the host to device.

**Kernel directive and Built-in variables**

The kernel syntax has remained the same as the C for CUDA syntax. The built-in variables are declared in the same way as in C for CUDA except the components are accessed slightly differently. For example, what was threadIdx.x become threadIdx%x. Another built-in variable has been included to the collection, warpsize. This is an integer variable, which holds the value of the number of threads in a warp.

**Fortran intrinsics**

In addition to the syncthreads() function, some new intrinsic functions have been added to the API, which are available in device subroutines.

*gpu_time(clock)* Returns the clock cycle time value on the GPU.

*allthreads()* A single scalar logical argument is passed to the function. The compiler evaluates the arguments for all of the threads in that warp and assigns it the logical value .true. or .false.. The function value is only .true. if all threads in the warp are assigned .true. to the argument value. This is known as a warp-vote operation.

*anythread()* This function is another warp-vote operation, which performs in the same way as the *allthreads()* function, except the function is only .false. if all threads in the warp are assigned the .false. value to the argument.

**Runtime functions**

In CUDA Fortran, device memory is allocated by using the device attribute when declaring variables in the host code. However, device arrays can be dynamically allocated by the use of the allocate statement in the host code. Dynamically allocated device arrays without the save attribute are automatically deallocated on return of the subprogram. Alternatively, as CUDA Fortran interfaces with the CUDA runtime library, allocatable device arrays can allocate memory on the host using a call to the familiar, cudaMalloc(), function. In this eventuality, the device memory is freed using the cudaFree() function. In order to dynamically allocate pinned arrays, the allocate convention must be used. The success of allocating page-locked memory depends upon the availability of page-locked memory and the system being used. This can be checked by the use of an additional keyword in the allocate statement, pinned. A logical success value is returned.
Transferring data between the host and the device is much simpler in CUDA Fortran than the C for CUDA equivalent. Assignment statements are used to copy data from one memory space to the other. The variable on the right-hand side is the source, and the variable on the left-hand side is the destination. Data can be copied for device to host, host to device, device-to-device and host-to-host. This practice is also used to assign/copy data to an array/variable in constant memory. All assignments/copies are synchronous. The `cudaMemcpy()` function and its corresponding functions are still available for use through calls to the CUDA runtime library, for those who are more familiar with the C for CUDA conventions.

**Memory**

The memory hierarchy and management mores have been carried over from CUDA Fortran. These include the management of constant memory in host code and the management of shared memory in device code. However, in C for CUDA, the use of read-only textured cache is available through a collection of runtime functions. These enable the developer to bind data to a texture in host code, and then fetch data from texture memory space in the device code. These functions are absent from the CUDA Fortran API, hence reducing the number of options available to the developer when managing memory.

To demonstrate how the CUDA Fortran extensions and runtime functions are used, a simple vector-vector multiplication problem will be used as an example.

**Vector-vector multiplication example**

The best practices for porting code to GPUs with CUDA Fortran follow the same guidelines as when using C for CUDA. For this reason, this example will not be discussed in as much detail as the vector-vector multiplication example for the C for CUDA API. Instead, the main differences between the examples will be discussed along with any implications they may have on performance. The original code has not been included here as the calculation is the same as figure 2.4, the only difference is that it is written in Fortran.

*Figure 2.8* shows the host subroutine for the example. The arrays A, B and C are passed into the subroutine as assumed-shape arrays. The Adev, Bdev and Cdev arrays are declared on the device as allocatable arrays. Once the value of N is determined using the Fortran intrinsic `size`, the device arrays are allocated. Again, the blocksize has been chosen to be 32 so as to obtain the best performance. The data in the A and B arrays on the host is then copied over to the device using array syntax in a simple assignment statement. This example demonstrates two different expressions for copying data, which are both legal. The number of threads in a block and the number of blocks in a grid are declared using the, `dim3`, predefined type, in much the same way as in C for CUDA. The kernel invocation, thread synchronisation and deallocation of arrays happen in the same way as C for CUDA.
subroutine host( A, B, C )
    real, dimension(:) :: A, B, C
    real, device, allocatable, dimension(:) :: Adev,Bdev,Cdev
    integer :: N, blocksize
    type(dim3) :: dimGrid, dimBlock

    N = size( A, 1 )
    blocksize = 32

    allocate( Adev(N), Bdev(N), Cdev(N) )

    Adev = A(1:N)
    Bdev(:) = B(1:N)

    dimGrid = dim3( N/blocksize)
    dimBlock = dim3(blocksize )

    call mmul_kernel<<dimGrid,dimBlock>>>( Adev, Bdev, Cdev, N, blocksize )
    r = cudathreadasyncsynchronize()
    deallocate( Adev, Bdev, Cdev )
end subroutine host

Figure 2.8 Example host code for a vector-vector multiplication using CUDA Fortran.

attributes(global) subroutine vectormult( A, B, C, N, blocksize)
    real, device :: A(N), B(N), C(N)
    integer, value :: N, blocksize
    integer :: i, tx

    real, shared :: Asub(blocksize), Bsub(blocksize)
    real :: Csub

    tx = threadidx%x
    i = (blockidx%x*1 + tx

    Csub = 0.0
    Asub(tx) = A(i)
    Bsub(tx) = B(i)

    Csub = Asub(tx) * Bsub(tx)
    C(i) = Csub
end subroutine vectormult

Figure 2.9 Example device code for a vector-vector multiplication using CUDA Fortran.
Figure 2.9 demonstrates the device code for this example. The device subroutine is qualified using the global attribute. The arrays are declared on the device and the scalars are declared as value. This ensures that the actual value of the scalar is transferred when the kernel is called, eliminating the need for the scalars to be explicitly transferred. The sub-vectors Asub and Bsub are declared with the shared attribute. The built-in variables are used to calculate the relative indexing of the global and shared arrays for each thread. It is important to note that CUDA Fortran follows Fortran indexing conventions, so blockIdx and threadIdx start at 1, as opposed to 0 in C for CUDA. Each thread is responsible for one element in each array and computes the operation, placing the result into the scalar variable Csub, before copying it back to the global C array.

The optimisations carried out here are the same as in C for CUDA vector-vector multiplication example. However, CUDA Fortran has some more features that can be used when optimising ported code. These will be discussed further in chapter 5.

2.4.3 CUBLAS Library

The CUDA SDK Toolkit offers an implementation of BLAS that executes on the GPU. It is called CUBLAS and is built on top of the CUDA runtime library. It offers a selection of BLAS1, BLAS2 and BLAS3 functions, in both single and double precision. In order to use the CUBLAS functions, memory must be allocated on the GPU to hold the arrays involved in the function, and the data must be transferred from the host to the device, before any CUBLAS functions can be called. Once the function has completed, the developer is responsible for transferring the results back to the host from the GPU. The CUBLAS library contains functions to carry out these memory management operations. In this project, an investigation into using the CUBLAS library to port one of the BLAS functions is carried out. A detailed description of using the CUBLAS library is discussed in chapter 4 [8].

2.5 Code suitability for acceleration using a GPU

As GPUs were originally designed for applications involving graphics rendering, the lack of sophisticated hardware limits the number of applications suitable for acceleration using this specialised architecture. Only a selection of the entire code is usually ported to GPU, which is known as a kernel. The rest of the code is executed on the CPU as normal. Careful consideration of the desired code for porting is required before any development proceeds. Code that has been poorly assessed, with respect to the hardware, could see no performance improvement or even a performance decrease after porting to GPU, compared with running the entire code on a CPU. This chapter discusses some of the reasons why the suitability of code needs to be assessed before porting code to the GPU. This chapter references the CUDA Programming Guide [1] and the CUDA Best Practices Guide [9].

GPUs are made up of hundreds of SP cores. Each SP core acts as an individual thread. Threads are executed in parallel within a thread block on one multiprocessor. Once one thread block has executed and terminated, a new thread block is created. The multiprocessor manages the thread blocks using a SIMT
(single-instruction, multiple-thread) architecture. The threads are designed to be lightweight and are scheduled with no overhead, in order to exploit fine-grained parallelism. Due to the nature of the threads working concurrently within a thread block, it is beneficial to reduce the number of idle threads. This makes GPU acceleration suitable for highly parallel code, such as data parallel problems (problems where the dataset is divided between the processors, a.k.a. data decomposition), enabling the maximum performance to be achieved using the many-threaded architecture.

Memory access latency is a large overhead when porting to GPU, as the device cannot directly access memory on the host and vice versa. It has to be explicitly transferred. The bandwidth between the device and the host memory is a lot lower than the bandwidth between the device and the device memory. For example, the theoretical device bandwidth for the Fermi card is 134.11 GBs$^{-1}$, and the theoretical bandwidth between the host and the device is only 8 GBs$^{-1}$. As a result, transferring data between the host and the device can cause a bottleneck in performance. As the GPU replaces transistors dedicated to data caching and flow control with transistors dedicated to data processing, caching data which has been transferred from the CPU, cannot be used to hide memory access latency. This means that arithmetic intensive code is required in order to hide the memory access latency with calculations. Well-suited code will have a high number of calculations and a small amount of memory operations in order to see any improvements in performance, compared to running the code without the use of the co-processor.

Another method of hiding the memory access latency is to run code, which is of a large problem size. In order to reduce the start-up costs of allocating and freeing memory when transferring data from host to device, it is more beneficial to send one large data packet as opposed to several smaller data packets. This means the performance of code with a large problem size will be affected less by the start-up costs of host-device data transfer, as a proportion of the overall execution time, compared to code with a smaller problem size incurring the same start-up costs. This also suggests that codes, which reuse data elements, are appropriate for GPU acceleration as less data transfers between the host and the device will be required.

The CUDA programming model implements the concept of each element of data in a domain acting as an individual thread. The threads are organised into thread blocks, allowing the programmer to implement operations across domain elements. The thread blocks are then organised into grids of thread blocks and mapped onto the SMs in hardware. Each threadblock is executed independently from other thread blocks. All threads in a block are presumed to be executed on the same SM. Threadblock size is therefore limited by the amount of memory on an individual SM. The maximum number of threads allowed per block is 512. For small domains it may be feasible to map all the elements to just one threadblock. However, larger domains may be affected by this memory restriction, resulting in the use of multiple thread blocks. As thread blocks execute independently, there is no way of managing communication between thread blocks, as there is no guarantee of the order in which threads are executed within the threadblock. Threads are able to communicate within a threadblock through the use of shared memory and synchronisation. So,
only if the domain being operated on is mapped on to one single threadblock can
inter-data dependencies between data elements exist. As for most scientific codes,
this would be unfeasible due to the memory restriction of an individual SM, it is
necessary that codes have few (if any) inter-data dependencies between data
elements (threads) within domains.

Another admirable feature in code being ported to GPU is the ability to be able to
overlap the execution of code on both the host and the device. In order to get good
performance out of a piece of code, it is necessary to reduce the amount of time
processors spend being idle. If the CPU is waiting for kernels to be executed on the
GPU before it continues executing the program, the performance of the code will
suffer as, the host spends time being idle. Suitable code will allow the host to
offload kernels to the device and then carry on executing the program without the
need to wait for return results from the coprocessor.

In hardware, thread blocks are distributed across the SMs in warps. A warp is a
group of 32 consecutive threads. The SIMT unit issues instructions to the warp of
threads one at a time. The same instruction is issued to every thread within the warp
and then executed. The parallel nature of the threads within a warp is entirely
dependent on all threads executing the same instruction. This means that if the
execution path of threads within a warp is conflicting i.e. by a conditional statement,
the thread, which has branched off onto a different execution path, will serially
execute the path, disabling all other threads in the warp until the conditional branch
path has been executed, at which point all threads converge back to the same
execution path. Conditional branching can therefore be highly damaging to the
performance of a GPU as they force serial execution. For this reason it is advisory
to avoid the porting of code, which contains conditional statements, or to alter the
code such that they are avoided.

2.6 MUSE

MUSE is a program, which calculates the electronic structure of molecular systems.
It was developed at Manchester University from within the Chemistry department.
The code is written in the Fortran77 programming language and makes use of linear
algebra libraries such as NAG and LAPACK. The version of the program used for
this dissertation has been modified to enable its use as a stand-alone program, the
original program interfaces with the molecular structure calculation software,
Gaussian.

MUSE uses a semi-empirical approach to calculating the electronic structure. Semi-
empirical methods employ approximations to the calculation being carried out, by
correlating with existing data i.e. physical properties of the system such as ionization
energies. This is done to enable the treatment of larger systems, as carrying out the
full calculation would be too computationally expensive [10].

The electronic structure of a molecule is calculated using the self-consistent field
(SCF) algorithm within MUSE. Self-consistent field (aka Hartree-Fock) theory
enables the determination of the ground-state (lowest energy) wave function and
ground-state energy of an N-body system. Sets of N-coupled equations for the N-spin orbitals are derived. An iterative, fixed-point type algorithm is used to solve the resulting equations. The solutions are approximations of the exact wave functions and energies. The process is repeated until an error criterion is satisfied [11].

Within the iterative process there are calls to maths routines, which carry out matrix-matrix multiplications and matrix diagonalisations. In order to improve the time to convergence, a direct inversion of iterative subspace (DIIS) algorithm is used, which can half the number of SCF iterations required in most calculations [12].

In order for the program to calculate the electronic structure of a molecular system as it follows a reaction path (i.e. the minima and the transition state), it has been interfaced with geometry optimisation algorithms GAUSSIAN94 (G94) and GAUSSIAN98 (G98). These suites require a set of Cartesian forces and sometimes, Cartesian force constants. Numerical finite differences of energies and gradients are used to calculate these variables [12]. For a 1ns simulation with a time step of 1fs, 1million finite difference energy gradient evolutions would be required. The version of the code being developed during this project only performs one of these evolutions for convenience.

As MUSE employs techniques within the code to reduce the number of SCF iterations, the performance of the code for larger systems (i.e. over 1000 atoms) is much better than it’s alternative approaches such as density functional theory (DFT). However, there is one drawback to this method, results may be less accurate. By accelerating parts of the MUSE code using GPUs (Graphical Processing Units), it is hoped that the performance improvements gained will allow for the program to exploit a more accurate implementation of the algorithm it employs. This could improve the simulation of larger systems (>1000 atoms) both in accuracy and execution time [12].

AMBER is a code that has recently been ported to GPGPU with great success. Two simulations within the program were ported, the Generalized Born and PMEMD simulations. It was accelerated using the CUDA -enabled GPGPU and a speed-up of approximately 15 times faster than a single quad core CPU was reported [14]. AMBER is a similar program to MUSE and so, its success in being accelerated using GPU is one motivation for porting MUSE to GPU.
3 Benchmarking

In this chapter, two separate comparisons are being made. The first is the difference in performance of the Tesla C1060 GPU architecture and the Fermi GPU architecture. The second is the difference in performance between the C for CUDA API and the CUDA Fortran API. The performance differences when a simple matrix multiplication benchmark is run, will be discussed with respect to the above comparisons. The C for CUDA benchmark was provided in the CUDA SDK toolkit [15], and the CUDA Fortran benchmark was provided in the CUDA Fortran code samples [16]. Both benchmarks carry out the exact same operation, and have been optimised to use the same memory spaces. The matrix multiplication benchmark uses double precision, as this is the precision used in MUSE. These benchmarks will be used to compare the performance of MUSE on the different systems and using the two different APIs, after it has been ported and optimised to use the GPUs. The compiler used on both architectures, to compile the C for CUDA benchmark, was the NVIDIA CUDA compiler, nvcc, version 3.0. The C for CUDA and the CUDA Fortran matrix multiplication codes are described in detail in Appendix A and B respectively.

3.1 Benchmarking the architectures

The architectures being compared are the Tesla C1060 and the Fermi GPU architectures. They have compute capabilities 1.3 and 2.0 respectively. With different compute capabilities, comes different specification and features, which affect the performance of the GPU. In order to understand the performance differences when running the benchmark on the two different GPUs, it is important to have a good knowledge of the design of the architecture for each compute capability. The Tesla C1060 will be referred to as simply the C1060 in the remainder of this dissertation.

Architecture

The specifications of each multiprocessor in the C1060 have already been mentioned in section 2.1. The number of cores in the Fermi quadruples from the C1060 to 32 per multiprocessor. The effect of which, should increase the speed of execution for kernels that utilise more threads than there are cores available. The number of special function units and the number of warp schedulers both double, increasing to 4 and 2, respectively. The Fermi no longer requires any specialised double precision units, as the IEEE 754-2008 floating-point standard is used, which provides a fused multiply-add (FMA) instruction for double precision, as well as single precision. This can be faster than doing the multiply and add operations separately. Each multiprocessor can carry out a maximum of 16 FMA operations per clock cycle. A clock cycle is the rate of processing in the core [17].

The C1060 issues instructions from the warp scheduler to all the threads in a warp over 4-clock cycles for integer/single precision arithmetic; 32-clock cycles for double precision arithmetic; and 16-clock cycles for single precision transcendental.
As the Fermi GPU has 2 warp schedulers, each warp scheduler can only issue instructions to half the cores. One scheduler is responsible for cores with an odd ID and the other is responsible for the cores with an even ID. Each scheduler issues instructions for integer/single precision arithmetic to a warp over 2-clock cycles. The number of clock cycles it takes to issue a double precision arithmetic instruction has been dramatically improved to 2-clock cycles. This should improve the execution time of kernels that contain double precision arithmetic operations for the Fermi, compared with the C1060. It is important to note however, the action of a warp scheduler issuing a double precision arithmetic instruction inhibits the other warp scheduler from issuing an instruction. Single precision transcendental instructions are issued to a warp in 8-clock cycles, half the amount it takes for the C1060 [18].

In both GPUs there is a read-only constant memory cache, accessible to all cores, which speeds-up reads from the constant memory space in device memory. The access to the texture cache is slightly different for each GPU. A texture unit, which speeds up reads from the texture memory space in the device memory, accesses the texture cache. In the C1060, multiprocessors are grouped into 3’s, and each group of 3 shares the read-only texture cache. In the Fermi, each multiprocessor has its own read-only texture cache. As a result of this, the performance of the Fermi would be expected to be better than that of the C1060 for codes that utilise texture memory, as their will be more chance of a texture cache hit occurring i.e. the value is in texture cache memory, due to less threads utilising each texture cache. One significant difference between the caching abilities of the two GPUs is that, the C1060 has no caching facility for local and global memory, whereas the Fermi does. Each multiprocessor has an L1 cache and there is an L2 cache shared by all.
Figure 3.1 Fermi memory hierarchy vs. Tesla C1060 memory hierarchy.

multiprocessors. These caches store memory accesses to global and local memory. This enables the performance of global and local memory accesses to be improved, as data is stored in the cache for future memory accesses. Register spills are also cached. The expected effect of this would be to increase the overall performance of all codes, as all data is initially placed in global memory when transferred from the host to the device. Figure 3.1 compares the memory hierarchy of the two different architectures [18].

Global memory

The way in which data is accessed in global memory differs between both GPUs. As discussed in the vector-vector multiplication example in section 2.4.1, a global memory request for a warp of threads is split into two individual requests when using the C1060. Coalesced global memory reads are crucial to ensure the best performance. Global memory is segmented into blocks. The block size varies from 32 bytes to 128 bytes depending on the precision of the data being accessed. Only one memory transaction can be carried out at a time. It is, therefore, beneficial if all threads are trying to access data in the same memory segment. Otherwise, the memory requests become serialised. There are no restrictions on the words that threads can access i.e. multiple threads can access the same word if necessary [18].

The memory hierarchy of the Fermi GPU is somewhat different due to the added L1 and L2 caches. Accesses to global memory are now cached. Each cache line is 128-bytes in both caches. This aligns with a 128-byte segment of device memory. The number of separate memory transactions issued when a warp requests memory, is dependent on the size of the words that the threads are trying to access. For 4-byte words, one memory request is issued per 128-byte segment. For 8-byte words, each half-warp issues a memory request and for 16-byte words each quarter-warp issues a memory request. Once the memory requests have been processed, they get broken down into cache line requests. The speed of these requests depends on whether there is a cache hit i.e. the value is in cache memory, or a cache miss i.e. the value is not in cache memory and must be fetched from global memory. In the event of a cache miss, the throughput is that of global memory [18].

Shared memory

As described in the vector-vector multiplication example in section 2.4.1, shared memory in the C1060 is organised into 16 banks which each store 32-bit words. The Fermi GPU’s shared memory is organised into 32 banks, which similarly assigns consecutive 32-bit words to consecutive banks. For both architectures, each bank can process 32-bits per 2 clock-cycles. As a result of having 32 banks, the Fermi GPU can be subject to bank conflicts between half-warps [18].

Results

A matrix multiplication example from the CUDA SDK Toolkit was used as a benchmark to obtain these results [15]. The code is written in the C programming
language and uses the C for CUDA extensions. The benchmark was executed on both the Fermi and the C1060 architectures. All execution times refer to the fastest time out of 5 runs. This is done to eliminate any background noise and any effects of other users running jobs on the system. The computation in the code was looped over 100 times to enable significant execution times on the GPU to be obtained, for comparison. The size of the matrices was chosen at 512 x 512 double precision elements. The number of threads per block was arbitrarily chosen as 16 x 16 in a 2-dimesional block, as this resulted in a multiple of 32, which prevents threads within a warp from being idle. The number of blocks was set at (512/16) x (512/16) in a 2-dimensional array of blocks. A full description of the code can be found in Appendix A.

Figure 3.2 Graph to show the fastest execution times of a matrix multiplication benchmark, out of 5 runs, on the C1060 and the Fermi architectures.

*Figure 3.2* shows the time it took to execute the matrix multiplication benchmark on both the Fermi and the C1060 architectures. Two separate timings were carried out. One that included the data transfers between the host and the device (with io), and one that excluded the data transfers between the host and the device (without io). This was done to expose the actual performance of the GPU.

As can be seen from the graph, the Fermi GPU out-performed the C1060 in executing the actual computation. The fastest execution time excluding the data transfers for the Fermi was 0.2842s, whereas for the C1060 it was 0.4463s. However, when the time taken to transfer the data was included in the execution time, the C1060 significantly out-performed the Fermi GPU. The fastest execution times were 0.5448s and 1.5845s, respectively.
**Analysis**

To fully explain these results, some details of the design of the matrix multiplication kernel, needs to be understood.

The matrices being operated on are two-dimensional matrices of size 512 x 512 filled with double precision elements. In the host code, matrices A and B are allocated on the host before being allocated on and transferred to the device. A third matrix, C, is allocated on the host and the device. The results of the multiplication are assigned to this matrix. After the kernel has been executed, matrix C is copied back to the host from the device.

![Figure 3.3 Diagram to illustrate the matrix multiplication code design.](image)

In the host code, the global matrices A and B are dissected into sub-matrices of size blocksize x blocksize. The blocksize is the number of threads in a block, 16. These sub-matrices reside in shared memory and each block is responsible for one sub-matrix of A and one sub-matrix of B. The sub-matrices are copied from the global matrices in global memory to the shared matrices in shared memory in a coalesced fashion. Each thread then computes one element of the sub-matrix, C, which entails loading in all the sub-matrices of A and B, required to compute the C sub-matrix, in a loop. Each threads has it’s own scalar variable, Csub, in which the answer is stored. Once the operation is complete, each thread is responsible for copying one result back to the global matrix in device memory. This is illustrated in *figure 3.3*. 

24
Kernel execution excluding data transfers

The execution time, excluding data transfers, for the Fermi is nearly half that for the C1060. This could be a consequence of the Fermi having double the number of cores per multiprocessor. This would mean that each multiprocessor could process double the amount of threads at once, hence halving the execution time. However, this would be presuming that all cores are active 100% of the time, which may not be the case. In this case, the number of blocks is 1024, which results in the execution of 73.14 blocks per multiprocessor on the Fermi GPU, and 34.13 blocks per multiprocessor on the C1060. Nonetheless, as the Fermi holds 4 times the number of cores per multiprocessor, the 73.14 blocks per multiprocessor will be processed 4 times faster than the 34.13 blocks per multiprocessor on the C1060. This results in an expected ratio of execution times of 1:1.86 for the Fermi. The actual ratio obtained was 1:1.57.

Another factor to consider is the rate at which instructions are issued from the warp scheduler. As only one warp scheduler can issue double precision arithmetic instructions at a time on the Fermi, double precision instructions are issued to an entire warp in 4-clock cycles. As the clock rate of the Fermi is 1.147 GHz, this means it takes 3.487x10⁻⁹ s to issue the instruction. It takes 32-clock cycles for the C1060 to issue a double precision instruction. It has a clock rate of 1.296 GHz, which works out at 2.468x10⁻⁸ s per issue. Both the higher clock rate and the improved rate of issuing instructions could be contributing factors to the Fermi outperforming the C1060.

The benchmark utilises the on-chip shared memory of the GPUs. As the Fermi card has caches available to store global memory accesses, this could significantly improve the performance from the C1060, as when the new submatrices are being loaded from global memory to shared memory, they may already be cached from the previous memory request. This means that the data copy to shared memory would have a throughput of that of the cache, rather than that of device memory, as is when using the C1060.

Finally, the C1060 shared memory is organised into 16 banks, which each hold 32-bit words. Double precision words are 64-bit. This means that in order to store data in the shared memory banks, each 64-bit word has to be split across 2 banks. As a half-warp constitutes 16 threads, this means there will be a two-way bank conflict for each thread when a shared memory request is being serviced. Hence, the bandwidth is decreased by a factor of 2. The Fermi shared memory however, organises its banks into 32, which each hold 32-bit words. As the size of a half-warp is still 16, there are no bank conflicts when accessing double precision data stored in shared memory. This, again, could be a cause of the Fermi’s quicker execution time.

Kernel execution including data transfers

Data is transferred from the host to the device along the same PCIe x16 Gen2 interconnect when using both GPUs. The same amount of data is transferred between the host and the device, and the same calls to the CUDA runtime library are
used to carry out the memory management functions. The results from the benchmark do not reflect this, as the total execution including the data transfers for the Fermi is 1.5945, which is nearly 3 times that of the C1060. These results are unexpected and could be a consequence of the set-up of the devices on Ness.

3.2 Benchmarking the APIs

The matrix multiplication example was used to benchmark the two APIs, C for CUDA and CUDA Fortran. This was carried out on the Fermi architecture, as the CUDA Fortran compiler was not available on the C1060 architecture. Again, the device code was looped over 100 times, in order to achieve significant execution times, for comparison. The time taken to execute the kernel was recorded, excluding data transfers, and including data transfers in order to expose the actual performance of the GPU. To compile the CUDA Fortran benchmark, the pgfortran 10.6 compiler was used [16] [21].

Results

The results show that C for CUDA benchmark out-performs the CUDA Fortran benchmark, marginally, for both the kernel execution including and excluding data transfers. The fastest execution time excluding data transfers was 0.2842s when using the C for CUDA API, and 0.3206s when using the CUDA Fortran API. The fastest execution time including data transfers was 1.5945s for C for CUDA, and 1.6447s for CUDA Fortran.

![Matrix Multiplication Benchmark](image)

**Figure 3.4** Graph to show the fastest execution times of a matrix multiplication benchmark, out of 5 runs, using the C for CUDA and the CUDA Fortran APIs, on the Fermi architecture.
Analysis

The matrix multiplication is implemented in the exact same way in both benchmarks. This leads to the conclusion that any performance differences are due to differences at compile time. The default setting of the PGI compiler enables level 1 compiler optimisations. These include scheduling of basic blocks and the allocation of registers [21]. No default optimisation settings appear to be present in the nvcc compiler manual. The results obtained are expected as the C for CUDA compiler is considerably more mature than the CUDA Fortran compiler. This means that the API has undergone more modifications to the original design to improve the performance of the compiler. The C for CUDA compiler used was version 3.0, many revisions have been made since the first version was released [15]. The CUDA Fortran compiler is built on top of the PGI 10.6 compiler, and is the first release [21]. These results will be used later in chapter 6 when analysing the execution times of the ported application in the two different APIs.
4 Porting

This chapter describes the stages of development that were undertaken in order to port MUSE to GPGPU. The port was undertaken using both C for CUDA and CUDA Fortran. Any differences between porting using the two different APIs will be described, where it is relevant. The porting process has been split up into four individual stages: identifying code and data for porting; converting to C (only relevant when using C for CUDA); decomposing into many threads; and debugging.

MUSE requires test-data to be input into the program. The test data used during development was that for a molecular system constituting 109 atoms. The main computation in MUSE was looped over 100 times and the average time taken, excluding input/output. This was done to ensure the timings were accurate. MUSE originally executed in 2.8824s. This time is the fastest time out of 5 runs. This is done in order to minimise any effects of other users on the system.

| Flat profile: |
| Each sample counts as 0.01 seconds. |
| % cumulative | self | self | total |
| time | seconds | seconds | calls | s/call | s/call | name |
| 30.54 | 63.24 | 63.24 | | | | |
| 14.17 | 92.58 | 29.34 | 1018 | 0.03 | 0.03 | matmul |
| 12.23 | 117.92 | 25.33 | | | | |
| 6.22 | 130.79 | 12.88 | 1025 | 0.01 | 0.01 | formd |
| 5.88 | 142.98 | 12.19 | | | | |
| 5.10 | 153.54 | 10.56 | | | | |
| 4.91 | 163.71 | 10.17 | 6027264 | 0.00 | 0.00 | fock2 |
| 3.45 | 170.86 | 7.15 | 6027264 | 0.00 | 0.00 | rotate |
| 3.04 | 177.16 | 6.31 | 6176519 | 0.00 | 0.00 | sqzero |
| 2.61 | 182.57 | 5.41 | 1018 | 0.01 | 0.03 | dodis |
| 1.44 | 185.54 | 2.97 | | | | |
| 1.37 | 188.39 | 2.85 | | | | |
| 1.25 | 190.98 | 2.59 | 6027264 | 0.00 | 0.00 | rsl |

Figure 4.1 Execution time of functions in MUSE which, used more than 1% of the total execution.

4.1 Identifying code and data for porting

The MUSE application comprises of 169 individual files. In order to highlight where the application spends the majority of the execution time, the program was executed using the gnu compiler, with the profiling tool, gprof on the HPC facility Ness. As the execution time of MUSE is already quite fast, the main computation of the code was looped over 100 times, excluding input/output, in order to increase the accuracy of the profile, and produce some significant timings for analysis. The execution time is small as the data set being used is for testing and developing purposes. A real dataset would be much larger. Also, the version of MUSE being used during the developing process was a stand-alone version. The results of the profile were verified by executing the program 5 times with the profiler. The same results were obtained on each occasion. Figure 4.1 shows the results of the profile. Only functions which, used more than 1% of the execution time, are shown.
Identifying code for porting

As can be seen from the profile, the majority of the execution time is spent in 3 functions: \textit{dlasr}; \textit{matmul}; \textit{dgemm}. The first of which, is a BLAS function, the second a complicated matrix multiplication function, and the third, another BLAS function. As CUDA now offers a CUDA-enabled BLAS library, CUBLAS, it was decided to use this to port the BLAS function, \textit{dgemm}. Unfortunately, the CUBLAS library does not provide all the functions available in the BLAS library. The \textit{dlasr} function has not yet been added to the CUBLAS library, and so could not be ported.

The \textit{matmul} subroutine was investigated further to see if the code was suitable for porting to GPU. As can be seen in figure 4.2, the subroutine contains 2 nested loops around a series of three 1-dimensional matrix multiplications, which all involve the same matrices. This conforms to the requirement that code being ported to GPU should exploit data re-use, to minimise the affects that transferring the data has on the execution time.

The subroutine, in its original state, involved a counter, \textit{ij}, when looping through the matrix multiplications. This would create inter-thread dependencies, however, the dependencies could be easily removed by modifying the code.

The subroutine is able to exploit data parallelism, as the matrices can be split up into many-threads. This quality is ideal for porting to GPU. Also, the problem size is quite large, i.e. >40000 elements for three out of the four matrices required to be transferred to the device. This means the effect of transferring data between the host and the device is reduced, compared with smaller problem sizes, as the data transfer overheads are fixed per data transfer, regardless of size.

In the subroutine, there is no conditional branching, a favourable characteristic for porting the \textit{matmul} subroutine. This is because branching causes serial execution of each thread participating in the branch, and so can hinder performance. Also, no other functions are called from the \textit{matmul} subroutine. This makes porting to the GPU much simpler, as only the data in the subroutine is required to be transferred to the device.

After investigating the suitability of the \textit{matmul} function, it was concluded that the function was suitable for porting to GPU, as it conformed to many of the attributes that suitable codes possess. The subroutine was taken out of the original file and placed in a C for CUDA/CUDA Fortran file with the file extension .cu/.CUF, respectively. The original file was then amended so as it called the \textit{matmul} function. Figure 4.2 shows the computation to be ported to the device.

Identifying data for porting

The next step in identifying suitable code was to distinguish the data objects, which would be required to be transferred to the GPU.

The \textit{matmul} function involves four matrices in total, three of which are used in the array operations, and one is used to store the results. All these matrices were required to be allocated on the device and undergo one data transfer between the
host and the device. Matrices $a$, $b$ and $ifact$ were transferred to the device, from the host before the kernel, and matrix $d$ was transferred to the host from the device after the kernel execution had completed. In C for CUDA, these were allocated on the device using the `cudaMalloc` function, and copied to/from the device using the `cudaMemcpy` function. In CUDA FORTRAN, the arrays were allocated on the device using the array declaration `device`, and data was transferred to/from the

![Fortran 77 code](image)

**Figure 4.2** Fortran 77 code, the computation part of the `matmul` function of MUSE.
device using assignment statements, as described in section 2.4.2. These operations are carried out in the host code, which resides in the same file as the device code function.

Other variables required by the device, are the array sizes and the scalar variable one. The array sizes are declared in the matmul function, which is not evident in figure 4.2, as the declarations have not been included. It was not essential that the scalar variable, sum, be transferred to the device, as it is only used as a temporary variable. This enabled the variable to be created and destroyed on the device itself, therefore reducing the amount of data transferred. The scalar variables were passed to the device as arguments in the kernel call. It is important to note, that the actual values of the variables are required to be transferred. As, in C an argument of a function can either be a pointer or a value, the actual values of the scalar variables can be transferred in C for CUDA. This is not the case in Fortran. Fortran subroutines/functions pass the argument as a pointer, and so it is essential to declare scalar variables, being passed to the device, with the value attribute in CUDA FORTRAN.

4.2 Converting to C

As MUSE is written in Fortran 77, the matmul function needs to be converted to the C programming language when using the C for CUDA API. This was carried out in stages so as to ensure no bugs were transferred to MUSE. The subroutine was isolated to run stand-alone from the application. The arrays were filled with dummy values, and the results of a number of runs with different dummy values were recorded. After the conversion process was completed, the results of the converted matmul function were verified against the results from the original Fortran 77 version of the subroutine. This section discusses the stages of converting code from Fortran to C.

The code was converted to C loop-by-loop, to ensure the correct output was obtained. The Fortran 77 do loops were substituted for the C for statement. The variables within the code were declared using the ANSI C convention. ANSI C is a C programming standard, published by the American National Standards Institute (ANSI) [19].

When converting the loops, an important consideration was the array indexing. In Fortran, array indices start at 1. In C, array indices start at 0. This meant it was necessary to adjust the loop indices, so that the loop didn’t proceed beyond the size of the array being looped over. Another difference between the way Fortran and C handle arrays, is that Fortran stores arrays in column-major order, whereas C stores arrays in row-major order. Figure 4.3 illustrates how arrays are stored in memory by the two APIs. As the arrays in the matmul subroutine are 1-dimensional vectors, the storage convention does not apply, therefore, no modifications were required to account for this. Figure 4.3 becomes more relevant later on in this section when the CUBLAS function, dgemm, is discussed.
A major difference between the Fortran and C programming language is the way arguments are passed to a function/subroutine. As discussed in section 4.1, Fortran passes the arguments as pointers to memory addresses, not by value. As the subroutine \textit{matmul} is part of a large collection of subroutines, it inevitably had to be linked to the rest of the application, by a call to the host C function from the Fortran 77 code. As the arguments in Fortran are essentially pointers to memory addresses, the C host function had to accommodate for this, by receiving the arguments as pointers.

Another issue, which had to be addressed, is case-sensitivity. C is case-sensitive, whereas Fortran is case-insensitive. Usually, the Fortran compiler will manage cases by converting all characters to lower-case during compilation. For this reason, the conversion of the \textit{matmul} function to C was all done in lower-case. In addition to this, the Fortran compiler may append a trailing underscore to function call in Fortran. A simple test for this was to compile versions of the C code with and without a trailing underscore after the host function declaration and link it to the Fortran calling function. This revealed that the Fortran compiler was indeed appending a trailing underscore to the function call. To ensure the compiler would recognise the C function, a trailing underscore was appended to the host function declaration [20]. \textit{Figure 4.4} & \textit{4.5} contains some pseudo code to demonstrate how the C routine was called from the Fortran code. As can be seen in \textit{figure 4.5}, the C for CUDA host function has a trailing underscore and receives the arguments as pointers. The function is declared to be \textit{external “C”} to ensure the compiler recognises it to be in the C programming language, not C++.
Once the function had been successfully converted to C, the next stage was to link the C function to the rest of the application. Firstly the files were compiled into objects, which could then be linked together to form the executable. The application produces output when executed which, was used to test the correctness of the code by comparing the output produced, after conversion, with the original output. This was done by extracting important values from both the original output and the new output, such as the final energy, and performing a simple diff operation on them. Correct results are the same as the original to $10^{-6}$ figures.
4.3 Decomposing into threads

Before the problem could be decomposed into many-threads, the code had to be modified in order to remove inter-thread dependencies. This involved removing the counter variable, \( ij \), and replacing it with code, which enabled each thread to calculate its own counter variable. Figure 4.6 shows the code that replaced the counter variable.

![Figure 4.6 C code replacement for \( ij \) counter.](image)

As can be seen from figure 4.6, the \( ij \) counter is replaced with a for loop. This sums up the current value of the \( i \) index and its predecessors. For example, if \( i=4 \), the for loop would calculate \( 4+3+2+1 \). The current value of the \( j \) loop index is then added to the result of the loop to obtain the index, \( ij \). Similar code replaces the \( ij \) counter in the CUDA Fortran version, taking into account the different array indexing convention.

The next step in the process was to decompose the problem into threads. This is a very important step with regards to performance. The more threads the problem can be decomposed into, the more parallelism can be achieved. However, other factors need to be taken into consideration when making this decision. For example, the number of kernel launches that will result. As each kernel launch carries an overhead with it, it is favourable to reduce the number of kernel launches. There is, however, zero-overhead in creating and destroying threads. As the \textit{matmul} code being ported is a series of nested loops, the code could naturally be decomposed into loop iterations. The options were to either decompose the outer loop into threads, which would entail one kernel call, or to decompose the inner-loops (loops 20, 30 and 40 in figure 4.2), which would entail 3 kernel calls. The former decomposition strategy was initially chosen, as this reduced the number of kernel launches, and maximised the amount of calculations for each thread.

The number of threads created from adopting this strategy was 292 for this problem size. The initial number of threads per block chosen was arbitrarily chosen to be 32, resulting in 10 blocks of threads.

4.4 CUBLAS

The next function in the application to be ported was the BLAS function, \textit{dgemm}. This was to be ported using the CUBLAS library. The CUBLAS library is designed to be as accessible as possible to all C and Fortran users. The functions themselves are implemented in the C-style, as they are built on top of the C for CUDA runtime
library. In order to make interfacing to Fortran codes simpler, CUBLAS stores multidimensional arrays in the Fortran column-major convention. Figure 4.3 illustrates the storage layout of elements in an array in Fortran. CUBLAS also retains the Fortran 1-based indexing. As discussed in section 4.2, the calling conventions between C and Fortran are somewhat different. These calling conventions can also differ from platform to platform. For example, case capitalisation and appending a trailing underscore on function declarations. To assist Fortran programmers with these issues when using the CUBLAS library, CUBLAS provide wrapper functions, which are written in C. There are two options to choose from, one that doesn’t require any further modification to the code (fortran_thunking.c) and one that does (fortran.c). In choosing the latter, the programmer is responsible for allocating and deallocating device memory, along with transferring data between the host and the device (CUBLAS supplies functions for these operations in the runtime library). This however, will provide the better performance, as the thunking wrappers carry significant overheads. CUBLAS also provide macro’s, which can be used to compute an element in a multidimensional matrix when interfacing with the library from Fortran [8].

As can be seen in figure 4.1, when profiling code that contains calls to functions in the LAPACK/BLAS libraries, no information of the calling function is recorded. On further investigation of the dgemm BLAS call in MUSE, it was discovered that the call was being made from within another BLAS call, dgelss. This, unfortunately, meant that the dgemm function could not be ported to GPU, as the CUBLAS library does not support the BLAS function dgelss.

4.5 Debugging

In order to debug the code during development, the code can be compiled to use emulation mode by adding the flags -deviceemu/-Mcuda=emu for C for CUDA/CUDA Fortran. This is where the code runs solely on the host, emulating the execution of the device, allowing host debuggers to be used, along with print/write statements. However, the use of the emulation mode should not be taken to be exact. In emulation mode, the device code has access to the host code’s memory, and the intrinsic functions found on the host are available to be used. Also, each threadblock may be executed serially. This prevents certain errors/bugs in the code from being identified, such as race conditions.

The CUDA debugger has recently been released, CUDA-GDB [15]. This is a debugging tool, which allows the device code to be debugged whilst being executed on the device. Unfortunately, the debugger requires non-cuda code to be compiled with the gnu compiler. This was not possible during this MSc project, as the GNU Fortran compiler is not supported on either of the hosts of the GPU systems C1060 and Fermi.
5 Optimisation

After the initial port of the matmul function, the performance of the code actually decreased. An increase in the execution time when porting codes to GPU is not unusual behaviour, as optimisation is required in order to get the best performance out of the GPU. This chapter discusses the optimisation methods applied to the initial port of the matmul function. Firstly, the optimisation of the C for CUDA code on the C1060 will be described. Secondly, the optimisation process for the CUDA FORTRAN code on the Fermi GPU will be detailed. The optimised C for CUDA code which, was developed on the C1060, was also executed on the Fermi GPU. A comparison of the performance of the same code on the two different architectures is detailed in chapter 6.1.

5.1 C for CUDA optimisation on the C1060

The matmul function is called 14 times during the execution of MUSE. The timings displayed in this section will show the execution time for each one of these calls on running the application. The time spent by the program in the matmul function for the original code, when executed on 1 CPU on the C1060, can be seen in figure 5.1, along with the execution timings, excluding the data transfers between the host and the device, after the initial port. The times shown are the fastest times out of 5 runs. This was done to minimise any interference from other users on the system. The compiler used was the PGI version 10.6 compiler, along with the NVIDIA CUDA compiler, nvcc version 3.0.

![Matmul Function Execution Times](image)

Figure 5.1 Graph to show the execution times of the matmul function before and after porting to GPU (excluding data transfers).

The average time taken by the original code to execute the matmul function was 0.03285s in one run of MUSE. As can be seen in figure 5.1, the results for each call to the function when run on the CPU show some discrepancy between calls. This
could due to a lack of cache coherence i.e. requested data being available in the cache, in calls 10 and 11. It is certainly not due to a change in the workload between calls, as on each repeated execution of the program, the discrepancy exists for different call numbers. After porting the *matmul* function to the C1060, the average time became 0.06777s. The average execution time of the *matmul* function more than doubles, even when data transfers are excluded. This could be because all the reads/writes are from/to global memory. As the code stands (unoptimised) no memory management has taken place i.e. utilising shared memory. The accesses to arrays in global memory are irregular, which results in little, if any, coalescing of global memory reads/writes. This may be the cause of the increase in execution time, excluding data transfers, as global memory request, essentially, become serialised. The CPU has caching facilities, which store unused memory loads; these can be accessed in future memory requests. Therefore, a memory request, which results in a cache hit, has the throughput of the cache, as opposed to the CPU memory. The C1060 has no such caching facility, so every read/write from/to global memory has the throughput of the device memory, if no coalescing is achieved.

*Figure 5.2* shows the execution times of the *matmul* function after porting, including the time taken to transfer data between the host and the device, compared with the original time taken to execute the *matmul* function. As can be seen, the first call to the *matmul* function, after porting, takes significantly longer than the first call before porting. This is a direct result of transferring data between the host and the device.

![Matmul Function Execution Times](image)

**Figure 5.2** Graph to show the execution times of the *matmul* function before and after porting to GPU (including data transfers).

The theoretical bandwidth of data transfers between the host and the device for the C1060 is 8 GBs$^1$. To calculate the effective bandwidth, the equation below is used.

\[
\text{Effective bandwidth} = \frac{(R_b + W_b) \times 10^9}{\text{time}}
\]
$R_b$ represents the number of bytes a kernel reads from memory and $W_b$ represents the number of bytes written to memory [9]. The first call to the *matmul* function has an effective bandwidth of $8.694 \times 10^{-4}$ GBs$^{-1}$ for the data transfer, however, the second call to the *matmul* function has an effective bandwidth of $1.762$ GBs$^{-1}$ for the data transfer. There is a dramatic improvement between calls, although the bandwidth is still much lower than the theoretical bandwidth. One way to increase this bandwidth is to increase the amount of data being transferred per transfer. As the test dataset being used was the largest available for the project, the effect of increasing the amount of data per data transfer could not be investigated. It is quite unclear as to why only the first call to the *matmul* function sees the affect of the data transfer. The same instructions are executed on each call to the function. This includes the memory allocations and data transfers. One explanation could be that the compiler changes the code so as to reduce the amount of data transfers required.

The first stage in the optimisation process was to reduce the number of global memory requests. C for CUDA allows the programmer to manage the location of data in memory. The available memory spaces are global memory, shared memory, constant memory and texture memory. The C1060 has 16KB of shared memory per threadblock, 65.536KB of constant memory and 256 bytes of texture cache. As the accesses to the arrays within the code are irregular, each array in its entirety is required by each threadblock.

The first arrays to be addressed were the arrays $a$ and $b$ (as seen in *figure 4.2*), as these are responsible for 6 out of the 10 reads to arrays in global memory. The extent of both of these arrays is 42778 double precision elements. This amounts to 68.4448KB, too large to be placed in shared or constant memory. For this reason it was decided to attach the arrays to texture memory.

Texture memory is a read-only memory space that is cached. Data is bound to the texture using the runtime function `cudaBindTexture()` in the host code. The texture reference has to be declared at file scope, as some of the features of a texture reference are required to be constant and so, must be known at compile time. The textures are then accessed in the device code. In this case, the runtime function, `tex1Dfetch()`, is used, as the textures are allocated to linear memory. If a multidimensional array is being bound to a texture, the texture can be bound to CUDA arrays, instead of linear memory. Appendix C explains the details of CUDA arrays and how they can be used. The type of data returned by texture memory is limited to integer and single precision floating-point types. However, the use of the built-in vector types is permitted. This meant that in order to use the texture memory, the double precision elements had to be fetched using the `int2` built-in vector type. This stores the 64-bit double precision value in two separate, 32-bit, parts within the vector. This can then be converted back to double precision using the `__hiloint2double` feature [1].

The results of using textured memory can be seen in *Figure 5.3*. The optimisation reduced the execution time of the kernel from the initial port. The average execution time of the kernel excluding data transfers became 0.05111s from 0.06777s. The performance is still worse than the performance when executing the kernel on the
CPU however. This could be due to memory requests experiencing cache misses when fetching data from texture memory space. The throughput of fetching data from texture memory is that of the texture cache in the event of a cache hit, and that of device memory in the event of a cache miss [9]. As the texture cache alignment is 256 bytes, this means only 32 double precision elements can reside in texture memory at a time. In the C1060, three multiprocessors share one texture cache. As

![Performance Comparison](image)

**Figure 5.3** Graph to show the execution times of the *matmul* function: before porting; after the initial port; after the use of texture memory (arrays a and b); and after the use of shared memory (array d) excluding data transfers.

there are 16 threads per block, presuming each threadblock resides on a different multiprocessor, 48 threads share one texture cache. This results in a maximum of 32 cache hits out of 48. However, there may be more than one threadblock per multiprocessor, also, as the array accesses are irregular, the data required by each thread may not be in the texture cache.

Another reason for the lack of performance improvement could be the overheads associated with converting between type int2 and double. This requires function calls, which carry overheads. These overheads may be quite minute on their own, but when multiplied by the number of texture fetches involved per thread, this figure could easily escalate. To test this theory, a program was written and executed on the C1060. This involved testing the built-in vector type’s performance against the performance of the same code when built-in vector types are not used. The built-in vector types used were __double2loint(), __double2hiint() and hiloint2double. The code carried out the matrix multiplication loop 20, as seen in figure 4.2. The device code that used the built-in vector types converted the double precision array elements to two integer values, and then converting the integer values back to a double precision values, before the matrix multiplication operation was carried out on each thread. The device code, which did not utilise the built-in vector types, simply carried out the matrix multiplication operation on the double precision elements in the arrays without converting between types. The full test code can be
found in the submitted source code. The results from this test can be found in table 5.1.

<table>
<thead>
<tr>
<th>Run number</th>
<th>Execution time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Built-in vector</td>
</tr>
<tr>
<td>1</td>
<td>0.006115</td>
</tr>
<tr>
<td>2</td>
<td>0.006118</td>
</tr>
<tr>
<td>3</td>
<td>0.006114</td>
</tr>
<tr>
<td>4</td>
<td>0.006117</td>
</tr>
<tr>
<td>5</td>
<td>0.006122</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>0.0061172</strong></td>
</tr>
</tbody>
</table>

Table 5.1 Results of the experiment to test the effect of the overheads when using built-in vector types.

As can be seen in the table, the average time for both device codes were very similar. Hence, the theory, that the overheads involved in converting between data types are responsible for the poor performance of the matmul function, is disproved.

As the d array is involved in one read and one write, the only memory spaces it could be stored in were global memory and shared memory, as the others are read-only. The next optimisation strategy was to store the d array in shared memory. The amount of shared memory available was not large enough to accommodate the entire array, and so the d array was dynamically stored in shared memory. Every 2 iterations of the j loop, threads would load the next 2 elements of the d array into shared memory, after placing the results from the previous 2 iterations back into the global d array. The aim of doing this was to reduce the time spent accessing global memory by a factor of 2, as double the amount of data would be being transferred at a time. This however was not the case, as can be seen from figure 5.3. The result was that the performance decreased from the use of texture memory to an average of 0.05724s excluding data transfers. The reason behind this could be that the shared memory accesses are suffering from bank conflicts. This is because the elements are of double precision floating-point type (64-bit) and the banks store words of size 32-bit. As there are only 16 banks in the C1060, this implies that there will be a 2-way bank conflict per shared memory access. This optimisation strategy was deemed unsuccessful.

The next stage in the optimisation process was to reduce the number of data transfers and also the amount of data being transferred. In MUSE, the matmul function is called twice from the file dodiis.F, successively. The same variables and arrays are used, except what was the a array in the first call is switched to the b array, and vice versa. In an attempt to reduce the data transfer overheads, the two calls were merged together, to form one function. In theory, this would reduce the time spent transferring data between the host and device, as half the number of data transfers would be taking place. Also, to reduce the amount of data being transferred per call, intermediate variables were created and destroyed on the device itself. Figure 5.4 shows a comparison between the execution time for both calls in the original code, the execution time for both calls (including data transfers) in the...
code optimised to use textures, and the execution time for both calls (including data transfers) combined to reduce data transfers. The latter also uses textured memory.

![Performance Comparison](image)

**Figure 5.4** Graph to show the execution times of the `matmul` function: before porting; after the use of texture memory (arrays a and b); and after reducing data transfers, all including data transfers.

As can be seen from *figure 5.4*, the efforts made to reduce the data transfer costs failed. One reason for this is that the compiler was already optimising the code to reduce data transfer overheads. This is evident in all results, as the first call to the `matmul` function takes the longest to execute. The results are very similar to that of the solely texture optimised code. The average time taken to execute the combined `matmul` function (including data transfers) in the code optimised to use texture memory is 0.27242s, whereas for the merged `matmul` functions, with half the number of data transfers, the average is 0.28684s.

The `ifact` array in MUSE is of size 292 integer elements. The entire array is required by each threadblock. As the size of an integer is 4 bytes, the total number of memory required by the `ifact` array is 1168 bytes (1.1684KB), which is small enough to fit into shared memory. The resultant execution times (excluding data transfers) after carrying out this optimisation can be seen in *figure 5.5* (Shared ifact series). The graph displays the execution time after optimising the code to use textures (excluding data transfers), to enable the performance to be compared. A decline in the performance of the code was seen after putting the `ifact` array in shared memory, when using 16 threads per block. After experimenting with different numbers of threads per block, the results clearly showed a performance improvement after changing the number of threads to 3. Details of the execution times obtained when varying the number of threads per block, can be found in Appendix D. It is highly
likely, that when there were 16 threads in a block, as there are 16 banks in shared memory, and the accesses to the ifact array are irregular, shared memory requests were subject to bank conflicts. A bank conflict only occurs when two or more threads are trying to access a different 32-bit word in the same bank [18]. So, when the number of threads per block is reduced to 3, the probability of multiple threads requesting data from the same shared memory bank is drastically reduced, hence the code performed better. The improvement in performance when using shared memory, as opposed to texture memory, is expected. The ifact array is no longer bound to a texture, and so has the latency of shared memory, which is approximately 100 times lower than a cache miss when fetching data bound to a texture.

![Performance Comparison](image)

**Figure 5.5** Graph to show the execution times of the matmul function: before porting-original; after the use of shared memory (array ifact), 16 threads per block-shared ifact 16; after the use of texture memory (arrays a and b)-texture; after the use of shared memory (array ifact), 3 threads per block-shared ifact 3, all excluding data transfers.

After optimisation, the matmul function utilised shared memory, texture memory and had a block size of 3. As can be seen from figure 5.5, the execution time, even excluding data transfers, was still higher than running the function on a CPU for this decomposition strategy. This is thought to be due to the irregular behaviour of the code when accessing arrays preventing parallelisation.

As an experiment, an alternative decomposition strategy was implemented. The function was split into 3 separate kernels, one for each loop that calculated the variable sum. The original thread decomposition culminated in a severely unbalanced load. For example, thread 0 would only execute 1 iteration of the $j$-loop, whereas thread 291 would carry out 292 iterations of the $j$-loop. The new thread decomposition culminated in each thread being load balanced and could also mean global reads/writes were coalesced more so, than the in original thread decomposition. However, the rest of the code was executed on the host. This meant that the outer loops ($i$ and $j$), looped around the 3 separate kernel calls. This greatly
increased the number of kernel calls within the function. As kernel calls have an associated overhead, this masked any performance improvements gained. The average execution time of the \textit{matmul} function, per call, during one run of MUSE, including data transfers was 2.857652s and excluding data transfers was 2.760384s. As a result, this decomposition strategy was rejected.

\section*{5.2 CUDA FORTRAN optimisation on the Fermi}

This section describes the optimisation process for the \textit{matmul} function ported to the Fermi GPU in CUDA FORTRAN. The compiler used was the PGI version 10.6 compiler, \texttt{pgfortran}. The times shown are the fastest times out of 5 runs. This was done to minimise any interference from other users on the system. The \textit{matmul} function is called twice consecutively in the code from a file called \texttt{dodiis.F}. This file is called 7 times, so the timings shown in \textit{figures} 5.6 and 5.7 show the execution time for both these calls, hence why there are only 7 calls as opposed to 14. This is done to allow a direct comparison to be made between all the stages of optimisation. After experimentation with the number of threads per block, similarly to the code development on the C1060, the execution time was found to be faster with a block size of 3 threads.

The execution times including data transfers are displayed in \textit{figure} 5.6. As can be seen, only the first two consecutive calls to the \textit{matmul} function show evidence of a performance decrease due to data transfers. This, similarly to the development on the C1060, is thought to be due to compiler optimisations. The addition of the data transfer overheads after the initial port causes the execution time of the first call to

![Performance Comparison](image)

\textbf{Figure 5.6} Graph to show the execution times of the \textit{matmul} function: before porting—original; after porting—initial port; after merging the two \textit{matmul} calls—merged calls; after putting the \textit{ifact} array into constant memory—constant memory; and after putting arrays \textit{a} and \textit{b} into shared memory—shared memory, all including data transfers.
increase to 2.38474s from the original time of 0.01893s. This is quite a substantial difference in performance and so, to reduce the time taken to transfer data between the host and the device, the consecutive kernel calls were merged together, resulting in half the number of data transfers during the execution of the entire code. The affect this optimisation had on performance can be seen in figure 5.6. The execution time including data transfers for the first two consecutive calls was reduced from 2.43233s, to 1.42000s when merged. This result however, is slightly puzzling, as it was only the first call to the matmul function that was experiencing an increased execution time due to data transfer costs in the first place. One theory behind this could be, that the restructuring of the code has enabled the compiler to optimise data transfers more efficiently. Hence, the reduction in the transfer time.

As can be seen in figure 5.7, the initial port to the Fermi resulted in a performance decrease, excluding data transfers. The average time taken to execute the matmul function on the GPU was 0.04673 s (excluding data transfers), where as the average time taken to execute the function on the CPU was 0.01892s. Even though the Fermi GPU offers an L1 cache and an L2 cache, the irregular behaviour of the global memory accesses could mean that a cache miss is more frequent that a cache hit. This could be the reason why the execution time on the GPU is much slower than that on the CPU, as the accesses to global memory are being serialised. Accesses to global memory on the GPU have a higher latency than accesses to memory on the CPU, if global memory accesses are not coalesced, as the CPU has an efficient caching hierarchy.

![Performance Comparison](image)

**Figure 5.7** Graph to show the execution times of the matmul function: before porting-original; after porting-initial port; after merging the two matmul calls-merged calls; after putting the ifact array into constant memory- constant memory; and after putting arrays a and b into shared memory- shared memory, all excluding data transfers.

The next stage in the optimisation process was to reorganise data in memory. The Fermi offers 65536 bytes of constant memory. As the ifact array is of size 292
integers and is only read in the code, it was decided to place this array into constant
memory. The results of this optimisation can be seen in figure 5.7. The use of
constant memory dramatically increased the performance of the kernel from the
initial port. The average execution time when using constant memory became
0.05333s (excluding data transfers) for the merged kernel, a significant reduction
from the average of 0.09347s (excluding data transfers) for the two consecutive calls
after the initial port. This is because, accesses to constant memory are usually
quicker than those to global memory [7]. Constant memory actually resides in a
section of global memory limited to 65536 bytes. The reason why constant memory
accesses are quicker than global memory accesses is due to the read-only constant
cache. By enabling the \textit{ifact} array to utilise the constant cache, the availability of the
L1 and L2 cache lines for global memory is increased, which could result in more
cache hits when accessing the arrays \(a\), \(b\) and \(d\) in global memory.

The final stage in the optimisation process was to test the performance when arrays \(a\)
and \(b\) were relocated to shared memory. The available shared memory in the Fermi
is 49152 bytes, too small to hold the entire arrays. The arrays were dynamically
copied to shared memory in sections, as each section was required by each thread
during runtime. The resultant execution times, excluding data transfers, can be seen
in figure 5.7. As can be seen, the execution time for each merged \textit{matmul} kernel
actually increased to an average of \(0.15000 \times 10^{-1}\) s. This is thought to be because,
when copying the sections of the arrays from global memory to shared memory,
coalescing is still not achieved. The reason for this being that each thread still
requires chunks of each array, preventing threads from successively reading from
global memory. For example, before the first loop that computes the \textit{sum} variable, if
the value of \(j\) is arbitrarily chosen to be 1, thread 2 in block 2 will copy 1 element of
the \(a\) array into shared memory, starting from the position set by the look-up table
\textit{ifact}(5). Thread 3 in block 2 will also copy 1 element of the \(a\) array into shared
memory, but this will be in the position set by \textit{ifact}(6). In order to ensure global
memory accesses are coalesced, for double precision floating-point values, the
elements being accessed by threads in a block have to reside in the same 128-byte
section of global memory. So, as the values in the \textit{ifact} array are not known at
compile time, it is near impossible to ensure copies to shared memory from global
memory are coalesced, as element \(a(ifact(5))\) may be in a different 128-byte block of
global memory to \(a(ifact(6))\). So, if coalescing is not achieved when copying to
shared memory, the code still carries the same overheads from uncoalesced global
memory accesses. The code also has to then request data from shared memory, an
additional cost to the performance of the code, hence why the average execution
time actually increases when using shared memory. This optimisation strategy was
unsuccessful.

After optimisation, the \textit{matmul} function had been reconstructed to reduce the
number of data transfers and utilised the constant cache. The execution times, both
including and excluding data transfers, were still higher than the for original code
being executed on 1 CPU however. This is thought to be due to the irregular array
accesses, and shall be discussed further in section 6.3.
6 Analysis

In this chapter, the performance of the ported and optimised C for CUDA `matmul` function when executed on the two different architectures will be compared. Following this will be a discussion of the performance of the `matmul` function on the Fermi GPU using the two different APIs. Finally, the performance when executing the function on the GPUs will be compared to executing the function on a CPU.

6.1 Performance comparison of the GPU architectures

The benchmarking results described in section 3.1 showed that the Fermi architecture performed better than the C1060 architecture when excluding the time taken to transfer data, with a ratio of Fermi:C1060 execution times of 1:1.57. The Fermi architecture performed much worse however, when the data transfer overheads were included, with a ratio of Fermi:C1060 execution times of 2.91:1. Consequently, the expected results after executing the C for CUDA `matmul` function on both architectures should display similar ratios. The actual ratio (obtained from the execution times displayed in figure 6.1) excluding data transfers was Fermi:C1060 1:1.04, and the ratio obtained for the execution times including data transfers was Fermi:C1060 1.07:1. Although the Fermi still performs better when excluding data transfers, and worse when including data transfers, the margin of difference in performance is much smaller. However, even though the `matmul` function carries out matrix multiplications, it is in a different manner to the matrix

![Performance Comparison](image)

**Figure 6.1** Graph to show the average execution times of the `matmul` function: on the CPU- original; on the Fermi GPU in C for CUDA with data transfers- Fermi including xfers, and without data transfers- Fermi excluding xfer; on the C1060 GPU in C for CUDA with data transfers- C1060 including xfers, and without data transfers- C1060 excluding data xfers.
multiplication carried out in the benchmark. Therefore, although the benchmark acts as a good guideline to performance, it is inevitable that the ratios of performance will vary for different codes.

The reason for the improvement in performance on the Fermi over the C1060 could be due to the use of texture memory. The Fermi GPU is organised so that each multiprocessor has its own texture cache, whereas the C1060 facilitates groups of 3 multiprocessors with a shared texture cache. This architectural difference could imply that when executing the `matmul` function on the Fermi GPU, a texture cache hit is more likely to be experienced when fetching data from texture memory, as discussed in section 3.1. This would cause an increase in performance with the Fermi.

The number of threads executing on each GPU is 292. They are organised into blocks of size 3. This means that each on the Fermi, each multiprocessor has 6.95 blocks to execute, whereas the C1060 has 3.24 blocks to execute. However, as the Fermi has quadruple the number of cores per block, the Fermi multiprocessors will execute 6.95 blocks 4 x faster than the C1060 multiprocessors will execute 3.24 blocks. This means that for every one block that the C1060 executes, the Fermi can execute 1.86 blocks. This however, presumes that all threads are active for the entire kernel execution, which may not be the case, as the ratio of actual execution times suggests.

### 6.2 Performance comparison of the APIs

The benchmark results in section 3.2 can be used to estimate the expected ratio of results when porting the `matmul` function using the two different APIs. The ratios for the C for CUDA execution times against the CUDA FORTRAN execution times obtained from the benchmark were 1:1.13 excluding data transfers, and 1:1.03 including data transfers. This implies that the C for CUDA API performs marginally better than the CUDA Fortran API and the results from executing the `matmul` function ported using the two different APIs should reflect this. The actual ratios (obtained from the execution times in figure 6.2) conflict the benchmark ratios when executed on the Fermi architecture. The ratios are 1.20:1 when data transfers are included, and 1.56:1 when data transfers are excluded. The reason for this could be due to the use of different optimisation techniques. In the C for CUDA code, texture memory is utilised to store arrays `a` and `b`. This technique improved the performance of the code when it was being developed on the C1060, however, the comparison being made here is for the execution of both codes on the Fermi architecture. As the Fermi architecture contains a caching hierarchy, it may no longer be optimal to use texture memory. Another difference in the optimisation techniques is the use of constant memory to store the `ifact` array in the CUDA FORTRAN version. In the C for CUDA version, this array is placed in shared memory.
Figure 6.2 Graph to show the average execution times of the *matmul* function: on the CPU- original; on the Fermi GPU in C for CUDA with data transfers- C for CUDA including xfers, and without data transfers- C for CUDA excluding data xfers; on the Fermi GPU in CUDA Fortran with data transfers- CUDA Fortran including xfers, and without data transfers- CUDA Fortran excluding data xfers.

The process of storing an array in shared memory involves copying the data from global memory to shared memory. If no coalescing occurs when copying this data, the result is that no improvement in performance will be obtained, as global memory is serially accessed in the same way as it would be if the arrays did not utilise shared memory.

6.3 Performance comparison between the CPU and the GPU architectures

As can be seen in figures 6.1 and 6.2, no performance improvement was obtained over the original code executing on 1 CPU. The reason for this is thought to be due to the irregular nature of the array accesses.

The GPU operates as a many-threaded parallel processing unit. This means that to perform at its optimum, many-threads have to be active at any one time. As the *matmul* function involves irregular array accesses, the more threads that were in a block would mean an even slower execution time, as the accesses to memory became serialised. So, by having irregular array accesses, a restriction on the size of each thread block was enforced. This meant that the many-threaded architecture that is responsible for speeding up many ported codes, could not be utilised to its full potential. In addition to this, the amount of computation in the kernel was not large enough to mask the overheads incurred when transferring data between the host and the device, making the overall execution time significantly faster than that when executed on a single CPU in all eventualities. However, this may only be the case
for the test dataset used throughout development. For a larger dataset, a performance improvement may be seen when the function is executed on the GPU. The reason being, that the amount of computation would increase, hence hiding the data transfer latency overheads. Also the code would become more compute intensive, allowing the many-threaded architecture to be exploited. This theory, unfortunately, could not be tested, as a larger dataset was not available.
7 Conclusion

In this dissertation an investigation into accelerating the HPC application MUSE was carried out. The porting process was carried out on two architectures, NVIDIA Fermi and NVIDIA Tesla C1060 using the APIs, C for CUDA and CUDA FORTRAN and the performance compared. To ensure a fair performance comparison was accomplished, a benchmark was implemented on the architectures and APIs and the results analysed. This chapter summarises the findings of this MSc project and the work that will be carried out in the future as part of PhD research.

7.1 Summary of findings

The results after benchmarking a matrix multiplication problem on the architectures showed that the Fermi architecture performed better than the C1060 architecture, when excluding data transfer times, nevertheless, when including data transfer times, the performance of the C1060 prevailed.

The results after benchmarking a matrix multiplication problem comparing the APIs showed that the C for CUDA benchmark performed marginally better than the CUDA FORTRAN benchmark, when data transfer times were both included and excluded.

The HPC application MUSE, was profiled and analysed in order to select suitable code for porting. One subroutine, `matmul`, was identified and so ported to GPU in both APIs. When using the C for CUDA API, the code had to be converted to the C programming language. The C for CUDA code was developed on the C1060, whereas the CUDA FORTRAN code was developed on the Fermi. After porting, optimisations were carried out and the resultant performance analysed.

With the given test dataset, neither the C for CUDA nor the CUDA FORTRAN developments of the `matmul` function resulted in an improvement in performance when executed on both GPUs, compared with executing the function on a CPU. However, contradictory to the benchmarking results, the CUDA FORTRAN version out performed the C for CUDA version when executed on the Fermi GPU. Whereas, the performance of the C for CUDA version, on both architectures, was consistent with the results from the matrix multiplication benchmark.

The CUDA FORTRAN programming interface provides much simpler extensions and functions when compared with the C for CUDA programming interface. However, the documentation for both APIs is lacking detail for some of the more complicated functions/extensions in their arsenal. This caused some delays in the development process. Nevertheless, NVIDIA provide an excellent forum for developers to broach their issues amongst the GPGPU community. This proved invaluable in addressing problems that lacked detailed documentation.
7.2 Future work

As the code, which has been ported to GPU, is a stand-alone version from the original MUSE program, future work would concentrate on porting the entire stand-alone version in a single-threaded manner. This could then be integrated into MUSE, and executed in parallel by the many-threaded GPU architecture, to speed-up the simulation, as running large systems with MUSE is very computationally expensive. This approach was beyond the scope of this project, due to the lack of availability of the CUDA FORTRAN compiler in the first months of the project. The time spent developing was increased significantly due to the need to convert code to C. Future work will also include accessing the structure of the code, and restructuring where possible in an attempt to reduce the amount of irregular array accesses.

A significant performance improvement would allow larger molecules i.e. enzymes, to be studied. The MUSE program exists in order to reduce the computational expense that similar programs, which use different techniques i.e density functional theory (DFT), experience. However, the method used in MUSE can be less accurate. By accelerating the entire MUSE program using a GPU, the performance improvement gained will allow for the program to exploit a more accurate implementation of the algorithm it employs. This could improve the simulation of larger systems (>1000 atoms) both in accuracy and execution time. This will be the focus of future work.
Appendix A: C for CUDA Benchmark

In this appendix the C for CUDA matrix multiplication benchmark is explained. The code can be viewed in the coding package that has been submitted. The code is courtesy of NVIDIA and comes straight from the SDK code examples [15]. The code will first be explained in terms of what the host code effectuates, followed by the instructions carried out in the device code.

The host code first calls the standard library function `srand`, which sets the initial value of the seed for creating random numbers. Three arrays are used in the host code for the matrix multiplication, two containing the data to be multiplied together, and one to store the final results. The arrays are 2-dimensional and set to size 512 x 512 double precision elements. After the sizes have been declared, the arrays are allocated memory on the host using the C function, `malloc`. Next the arrays are filled with random numbers by a call to a user-defined function, which uses the C standard library function, `rand`. After the arrays have been initialised, the first timer starts, this times the total kernel time, including data transfers. The arrays are then allocated device memory and copied over to the device, using the synchronous `cudaMalloc` and the `cudaMemcpy` function, respectively. The number of threads in a lock is set to 16 x 16, and the number of blocks in a grid is set to (512/16) x (512/16). The second timer starts before the device kernel is invoked. After the kernel invocation, a call to `cudaThreadSynchronize` is made in order to ensure all threads have finished before executing the next operation. The second timer is then stopped, this time will represent the time spent in the device kernel, excluding the time spent transferring data. The results of the matrix multiplication are then copied back to the host memory from device memory using the `cudaMemcpy` function. The first timer is then stopped before the deallocation of arrays in both host and device memory, and the threads exited.

In the device code, the threads operate on blocks of the global matrix iteratively when calculating the results. Each thread is given a local identification and an identity in the sub-matrix. A sub-matrix of each global matrix of size 16 x 16, is loaded into shared memory, each thread fetching one element. To ensure all the elements have been loaded into global memory, synchronisation occurs before commencing with the calculation. The sub-matrices are then multiplied together; each thread is responsible for computing one element of sub-matrix. Synchronisation occurs again, before the next sub-matrix is loaded into shared memory and the process repeated. Once all the sub-matrices have been loaded iteratively, and the result calculated on each thread, each thread then copies its result back to the global results array.
Appendix B: CUDA FORTRAN Benchmark

In this appendix the CUDA Fortran matrix multiplication benchmark is explained. The code can be viewed in the coding package that has been submitted. The code is courtesy of PGI and comes straight from the described code example in the documentation [16]. The code will first be explained in terms of what the host code effectuates, followed by the instructions carried out in the device code.

The arrays to be multiplied are filled with random numbers and the results array is initialised to zero in the main program, and passed into the host subroutine. The arrays are of size 512 x 512 double precision elements. The first timer is started, before the arrays are allocated on the device using `allocate` statements. This timer measures the total time taken to execute the code on the device including data transfers. The arrays are then copied from host memory to device memory using synchronous assignment statements. The number of threads in a block is set to 16 x 16, and the number of blocks in the grid is set to (512/16) x (512/16). The second timer is started and the device kernel launched. This timer measures the time taken to execute the kernel, excluding data transfers. Once the kernel has been executed, all the threads synchronise. After synchronisation the second timer is stopped. The results are then copied back to the host from the device using a synchronous assignment statement. The first timer is then stopped, before the arrays on the device are deallocated.

In the device subroutine, sub-matrices are declared in shared memory of size 16 x 16. Each thread is given a local identification, and an identity in the sub-matrix. The sub-matrices are loaded into shared memory; each thread is responsible for loading one element of each sub-matrix. The matrix multiplication is carried out for that sub-matrix, each thread being responsible for computing one element. The sub-matrices are iteratively loaded into shared memory after computation of the previous sub-matrix has been carried out. To ensure all threads have finished calculating their elements for the current sub-matrix, synchronisation of threads is enforced. Once the process is complete, each thread loads the result back to global memory.
Appendix C: CUDA Arrays

CUDA arrays are a type of storage available for use in conjunction with texture memory. They are specifically optimised for texture fetching, and offer a memory layout in either 1-, 2- or 3-dimensions. The elements of a CUDA array are made up of either 1, 2 or 4 components. These may be signed or unsigned, 8-, 16- or 32-bit integers, 16-bit floats (only available when using the driver API), or 32-bit floats.

CUDA arrays are the favoured alternative to linear memory when using multidimensional arrays, as features within texture memory i.e. texture filtering, are only available when using CUDA arrays. CUDA arrays are only accessible from the device via texture fetches. The CUDA array can only be bound to a texture reference holding the same number of packed components. The CUDA arrays are only writeable from host code.
Appendix D: Block size execution times

In order to deduce the optimum number of threads per block, after optimisation of the C for CUDA `matmul` function on the C1060, the block size was experimented with. Figure D.1 shows the results from the experiment. The block sizes tested were powers of 2, from 2-64. Above 64 threads, the thread count was increased by 32, until the block size equalled 192. The thread numbers chosen roughly followed powers of 2, as the number of cores and the warpsize are both powers of 2.

![Block Size Performance Comparison](image)

**Figure D.1** Graph to show the average execution times of the `matmul` function excluding data transfers, with varying numbers of threads per block.

As can be seen from the results in figure D.1, the larger the size of the block, the slower the execution time. This is thought to be due to the irregular nature in which, arrays are accessed, causing a restriction on the number of threads able to operate simultaneously. The fastest times were achieved from the lower powers of 2, and so, the performance when using few threads per block was investigated.

**Figure D.2** shows the results when the `matmul` function was executed with 1, 2, 3, and 4 threads in a block. The results show that the optimum block size is 3. As this gave the quickest average execution time, 0.08084s out of 100 runs. This could be due to the number of threads being low enough to minimise the amount of serial global accesses, whilst also providing some parallelisation. The balance of memory access overheads and number of active threads is at its optimum when 3 threads are active per block.
Figure D.2 Graph to show the average execution times of the matmul function excluding data transfers for block sizes 1-4.
8 References


