GPU Acceleration of HPC Applications

Alan Richardson

August 21, 2009

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2009

Abstract

The use of Graphics Processing Units (GPUs) for non-graphical computations, also known as GPGPU, has generated a large amount of interest due to the promise of impressive performance gains. The process of accelerating existing HPC applications through the use of GPUs is investigated in this dissertation. Five candidate codes were initially considered – CENTORI optimisation library, FAMOUS, Ludwig, CHILD, and CSMP – of which the first three were ported to partially run on a GPU using Nvidia's CUDA language. This did not result in a significant speed-up, and in some cases made the codes execute more slowly. Further optimisation was performed on the Ludwig and CENTORI codes, producing speed-ups of up to 6.5 times and 12 times respectively. When the time taken to transfer the necessary memory between the CPU and GPU was also considered, the speed-up was reduced to at most 3.6 times for Ludwig and the CENTORI kernels ran slower than the original CPU versions.

Contents

List of Tables

List of Figures

List of Listings

Acknowledgements

I am very grateful to Dr. Alan Gray and Mr. Iain Bethune for their advice and supervision during this dissertation. I would also like to thank SAAS, the University of Edinburgh, and EPCC for providing funding for my MSc. For providing me with excellent facilities that enabled me to complete this project, I am thankful to the Distributed Computing group at Daresbury Laboratory and the Edinburgh Computing and Data Facility (ECDF).

Chapter 1

Introduction

Graphics Processing Units, or GPUs, are a type of processor designed specifically to perform the calculations necessary for computer graphics. A field of research that has emerged recently and grown very rapidly is the use of GPUs to accelerate non-graphics applications. This is frequently referred to as General Processing on GPUs (GPGPU). At the moment it is rare to read an issue of HPCWire that doesn't contain an article about the latest developments in the GPGPU field. All of the recent HPC conferences have had GPGPU workshops and presentations. Many of the major scientific software projects are investigating the possibility of adding GPU capability to their codes. It's not hard to see what is driving this surge of interest in GPUs, with many reports of speed-ups from 20 to 100 times being touted from these devices which are relatively cheap, and already present in many PCs. HPC software developers who have not yet experimented with GPUs are understandably keen to know whether their codes can obtain such performance improvements too, and this dissertation will look at the process of modifying such existing HPC applications to use GPUs.

Two key elements were integral in the development of the GPU architecture. First, and most importantly, a very large and lucrative market of computer game players who were, and continue to be, willing to pay for faster and more detailed graphics rendering. The second is that the algorithms used in graphics calculations are of a specific form, which allowed the architecture to be optimised for such operations. The combination of these allowed the creation of devices that differ quite substantially from traditional CPUs.

Most computer applications are unlikely to run faster on a GPU because they do not operate in the way that GPUs are optimised for. However, many HPC codes do possess these characteristics, and so this is the area where most GPGPU work is concentrated. Graphics calculations are typically characterised by large numbers of single precision floating point operations being performed on arrays of data. Another feature is that they are often embarrassingly parallel, which means that the calculation can be performed on each data element independently.

Several years ago when the GPGPU idea was first considered, writing GPGPU applications was a difficult task as it was necessary to reformulate code as a graphics calculation that GPUs were designed for. The release in 2007 of the CUDA language by GPU manufacturer Nvidia made the process of writing GPGPU code much easier. A key element of this is the well written and extensive documentation provided by Nvidia, such as the CUDA Programming Guide [25] which describes the GPU architecture as presented to GPGPU developers, and provides guidelines to achieve good performance.

In this dissertation a thorough introduction to the important issues involved in GPGPU development is presented in chapter 2. This includes a more detailed discussion of what codes are suitable for GPU execution (section 2.1), the CUDA programming language (section 2.3), and a simple example to demonstrate the porting and optimisation of a code on a GPU (section 2.3.1). The five applications that will be considered for GPU acceleration (CENTORI optimisation library, CHILD, CSMP, FAMOUS, and Ludwig) are presented in section 2.4. Chapter 3 describes the process of deciding which of these codes is likely to benefit from the use of a GPU, and the initial porting of these. As the aim of using a GPU is to improve the performance of applications, simply getting the code to run on a GPU is usually not a satisfactory final result. Optimisation is therefore an important topic, and the process by which it was performed is discussed in chapter 4 together with a presentation of the resulting performance. The methodology used to test that the GPU implementations produced sufficiently similar results to the original CPU versions is described in chapter 5. Finally, the Conclusion (chapter 6) contains an examination of whether the benefits obtained by the use of GPUs make them useful for HPC applications.

Chapter 2

Background

2.1 Codes suitable for GPU acceleration

Typically, entire applications do not run on a GPU – most of the code still executes on the CPU, while small sections that consume a large part of the runtime could be accelerated by running them on the GPU. This means that only part of the code (the part that is ported to the GPU) needs to conform to the model that suits GPUs for good performance. Functions that execute on a GPU are called ‘kernels’.

High computational intensity CPU memory cannot be accessed directly by the GPU, nor can the CPU directly reference data that is stored on the GPU. It is therefore necessary for the data that will be needed to be explicitly transferred before it is required, by a function call on the CPU. GPUs connect to the host computer through a PCI or PCI Express (PCIe) connection [7]. This enables data to be transferred between the CPU and GPU at up to 4GB/s in each direction simultaneously, although in reality it is not possible to achieve such speed as it is limited by factors such as the speed at which the data can be fetched from memory. This means that in certain situations it may be faster to perform an operation on the CPU than to incur the delay of moving data to the GPU and then back again, even if the actual computation is performed faster on the GPU. For this reason, only computations that involve a significant number of floating point operations per data element are likely to achieve a speed-up when implemented on a GPU as in other cases the time saved in performing the calculation faster is unlikely to offset the transfer time.

Memory reuse and large problem sizes In addition to the transfer time, there is also a significant overhead associated with allocating and freeing memory on the GPU. This further restricts the use of GPUs to only those codes where each element of memory is used multiple times, such as by calling the kernel in a loop. The time required to allocate and free memory on the GPU increases much more slowly with increasing data size than the time for transferring the data. This implies that the cost of the allocation and free operations as a percentage of the overall runtime will be higher for small problem sizes. The cost of GPU memory functions is investigated further in Appendix A.

Highly parallel GPUs use many-threaded programming. Threads are intended to be light-weight; each thread is often assigned a single array element. In contrast to standard practice in HPC programming, there are usually many more threads than processors in GPU programs. Groups of threads are created, execute, die, and another group is then created. This implies that many (usually most) of the threads will not be active at a given time. Codes that run on GPUs must therefore be highly parallel, so that they can be decomposed into hundreds or thousands of threads, which will allow the GPU's capabilities to be fully utilised. It is also important that threads do not require interaction with other threads, since there is usually no guarantee that different threads will be active simultaneously.

Low inter-thread communication To operate on the problem sizes that are typical in HPC applications, it is often necessary to combine the computing power of several GPUs. This can be achieved by using MPI to communicate between the CPUs that the GPUs are connected to. GPU servers are also available, such as the Tesla S1070 [26], which contain multiple GPUs in server rack housing. Even in this situation, however, it is still necessary to communicate via the CPUs. The importance of minimising inter-thread communication is even more important in the multi-GPU situation, as communicating between GPUs would require data to be copied from the source GPU to its controlling CPU, then sent via MPI to the destination GPU's CPU and then copied onto the GPU. As transferring data between the CPU and GPU is an expensive operation, performing this frequently would be likely to erase any performance gains obtained by using the GPUs.

Minimal conditional execution As will be discussed in section 2.2 on GPU architecture, each thread is not independent – it is part of a group of threads. If the threads in the group do not all perform the same operation, then the different operations must be serialised. For example, if half of the threads in the group do not perform the same operation as the other half, then half of the threads will be idle for each operation. Conditional statements should therefore be avoided to ensure maximum efficiency.

Combined CPU and GPU execution Once a kernel has been launched on a GPU, no further input is required from the CPU until the kernel completes. GPU kernels are therefore launched asynchronously from the host CPU, allowing the CPU to perform other work while the GPU is busy. Ideally, for an application to use the GPU efficiently, it should be possible for the CPU to do useful work while the GPU is busy.

2.2 Architecture

It is the unusual architecture of GPUs that make them interesting, and understanding it is crucial for obtaining good performance. This section will therefore discuss in detail the differences between the design of CPUs and GPUs. The terminology of Nvidia will be used. ATI's products have a similar architecture, but the naming is sometimes slightly different.

2.2.1 Main differences to CPUs

To maximise the performance of GPUs for graphics calculations, several features present on CPUs have been eliminated in favour of increased floating point computation power. A sophisticated caching mechanism was one such feature. Storing recently used data in a quickly accessible location is useful for many of the programs that CPUs were designed to run as often only a small amount of data is active at one time and is reused many times. Automating such a feature relieves programmers of much work but also denies them the ability to maximise performance by carefully managing the cache. Simple programmer-controlled caches are therefore a better match for GPUs, as they allow more transistors to be dedicated to useful computation instead of managing the cache, while also giving greater optimisation control to the programmer. Another feature on which GPUs compromise is the control unit. These issue instructions to the floating point units, telling them what to do in order to perform the operation. CPUs are required to be very versatile, handling many different types of computation well. In order to achieve this, many advanced techniques have been developed such as pipelining and out-of-order execution. As GPUs are only designed to be good at a very particular type of calculation, such techniques, which consume valuable transistors, are not needed. In fact, in graphics calculations the same operation is frequently applied to every element in an array. This means that the instruction issued to the floating point unit will be the same for many array elements. It is therefore possible to further reduce the number of transistors used on control units (and therefore allow more for floating point units without increasing the cost) by only having one control unit for several floating point units. This is similar to a SIMD (Single Instruction, Multiple Data) machine, in which a group of processors perform the same operations in lock-step, but on different elements of an array.

2.2.2 Processors and threads

A GPU contains several Streaming Multiprocessors (SMs), each of which contains eight Scalar Processor (SP) cores. There is only one control unit per SM, so only one instruction can be issued at a time, which in turn means that all of the SPs in each SM must perform the same operation simultaneously. In typical graphics calculations, the same operation is performed on a large array. The operation of the GPU is optimised for this, so the SPs repeat the same instruction four times (on different data) before waiting for another instruction from the control unit. This increases the efficiency of the SPs as it reduces the time they spend waiting for what will normally be a repetition of the same instruction from the control unit. There are cases, however, such as at problem domain boundaries, where an instruction should not be repeated four times by eight SPs, as, for example, this might result in some of the SPs trying to operate on data that doesn't exist. For these situations another feature exists which allows SPs to be disabled, so that they can remain idle instead of performing an operation that they are not supposed to. Even if only a single operation is to be performed on a single data element, the situation is the same: the instruction will be issued to all of the SPs, but all except one will be disabled at first. For the three repetitions of the instruction, all of the SPs will be disabled. This is an enormous waste of processing power, but such an operation would be very rare in a graphics code. Each operation performed by an SP corresponds to an operation in a thread. A group of 32 threads is called a 'warp'. If all of the threads in a warp perform the same operation simultaneously (which should occur unless a conditional statement has changed the execution path of some of the threads), then this will be the instruction issued by the control unit. The eight SPs will perform the instruction for the first eight threads in the warp, and then it will be repeated for the next eight threads, and so on until it has been performed for all 32 threads. The instruction for the next operation in the code that the threads are executing will then be issued by the control unit, and the same procedure will be repeated. If threads within a warp do not follow the same execution path, then a divergent warp will result, which requires the different branches to be performed sequentially. In this situation, when an SP is about the perform an operation that corresponds to a thread that is not executing, then it will be disabled as described above.

Threads are grouped into thread blocks (frequently shortened to 'blocks'), which are three dimensional arrays to allow convenient mapping of threads to data arrays. The first 32 threads in the block will form the first warp, and subsequent groups of 32 threads will also form warps. Blocks are arranged on a two dimensional grid. Blocks are allocated to multiprocessors, so all of the threads in a block must be processed by a single multiprocessor.

To cover memory access latencies, SPs can time-slice threads. This means that if a warp of threads is waiting for a memory transaction to complete, instead of being idle the SPs can switch to other threads and do useful work. This relies on there being sufficient resources available to support additional threads being active. Each SM only has a small amount of on-chip memory so it is possible that launching new threads while others are still active (although waiting for memory access) may exhaust the available on-chip memory. Another limitation is the total number of threads that can be active on an SM at any one time, regardless of whether there is sufficient memory available. If either of these prevent new threads being launched to cover memory latency, then the SPs will have to remain idle until the memory request is completed.

As single precision floating point accuracy is sufficient for most graphics applications, this is what GPUs have been designed for. It is not possible to perform double precision floating point operations using most GPUs currently available. With the advent of GPU usage for scientific computation, which frequently requires double precision floating point accuracy, several products have been released to enable this. One of the key features of the 'compute capability 1.3' range of Nvidia GPUs (discussed in section 2.2.4) was the inclusion of one double precision unit per SM. While this makes double precision possible, the peak performance obtainable with double precision is about one tenth of that with single precision. This is due to there being eight times more single precision units than double precision, and presumably the remainder is caused by double precision calculations taking longer. During this dissertation double precision was used, as discussed in Appendix E.1.

In Nvidia's terminology, a GPU is frequently referred to as the 'device', and the CPU that it is connected to as the 'host'. It is not possible to use more than one host to control a single device, but it is possible (although not recommended) to have more than one device connected to a single host.

2.2.3 Memory

GPUs contain several different types of memory. These are described below. The layout of the various memory spaces in relation to the Scalar Processors (SPs) and Streaming Multiprocessors (SMs) can be seen in Fig. 2.1.

Device Memory/Global Memory The main memory on a GPU, the equivalent of a CPU's RAM, is called Device Memory. On high-end GPUs (Tesla C1060) there is 4GB of Device Memory (in the form of 'GDDR3' memory). The most noticeable difference compared to CPU memory is its large bandwidth – currently around 100GB/s compared to 10GB/s for a CPU. As many HPC codes are memory-bound, this may seem like a very significant advantage, however it must be remembered that this bandwidth is shared by all of the GPU's SPs, and so it is probable that the problem will be exacerbated on a GPU. This memory is mainly used for Global Memory, a large read/write area for storage that can be accessed by every SP. Memory transfers between the host and device must go through the Device Memory, typically writing input and then reading results back from Global Memory. Threads running on the GPU can access data stored in Global Memory in the normal way, without needing a function call as with some other memory forms discussed below. A complication is that accesses to Global Memory will only be done in parallel within a warp if they conform to requirements known as 'coalescing'. One of these requirements is that the data elements accessed by a half-warp of threads (16) must be consecutive in memory, and the start of the block of memory must be a multiple of 16 elements from the beginning of the array. In older GPUs the data elements had to be accessed in sequence, so the first thread in the half-warp must access the first data element in the block, the second must access the second, and so on. This condition has been relaxed on the latest products, and it is now possible to have accesses in any order within the data block, or even to have multiple threads

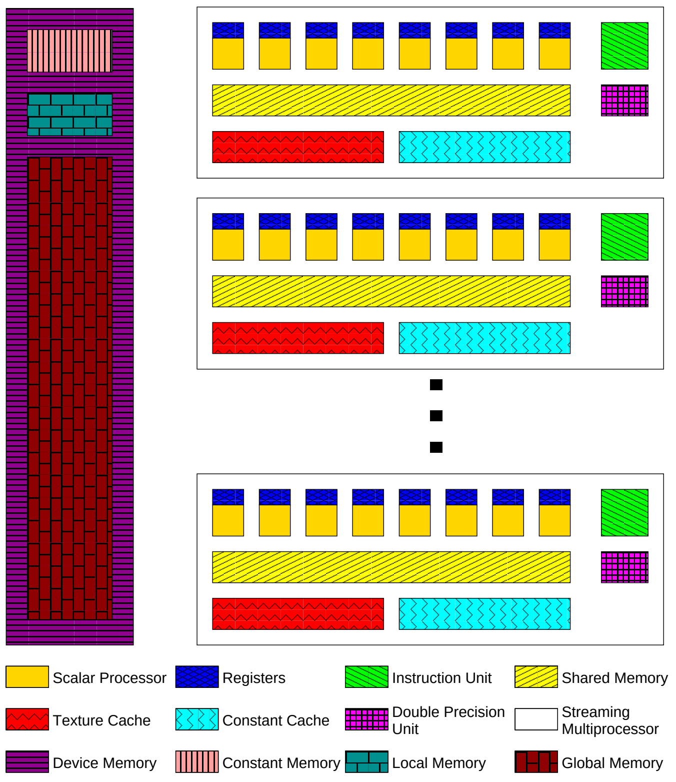


Figure 2.1: A diagram of the basic GPU architecture as exposed to CUDA developers

reading the same element. If coalescing is not achieved, then the memory accesses by the threads in the half-warp will be serialised. The advantage of achieving coalesced memory access is visible in Fig. 2.2. Another issue with Global Memory is referred to as 'partition camping'. This arises from the fact that the memory is arranged in banks. Simultaneous accesses to the same bank, even if the access is coalesced, must be serialised.

Shared Memory The second most important form of memory on a GPU is the Shared Memory. This is a small on-chip memory, 16KB per SM on current products, but which only takes about one cycle to access. It is, in effect, a programmer-controlled cache. Employing this memory is very important for achieving good performance on a GPU as it is not possible to avoid having a memory-bound program without it. The strategy encouraged by Nvidia suggests that data should be read from Global Memory into Shared Memory, operations performed on the data (preferably re-using the data several times), and then written-back to Global Memory. The most prominent characteristic of Shared Memory is that it is formed of sixteen banks. The data is distributed so that consecutive words are on different banks. Each thread in a half-warp must access a separate bank or a bank conflict will arise which results in access being serialised for the conflicting accesses. If every thread in the half-warp accesses a single element, however, then a broadcast will be used to avoid serialisation. While it is probable that the data from several blocks will be present in an SM's Shared Memory at one time (due to

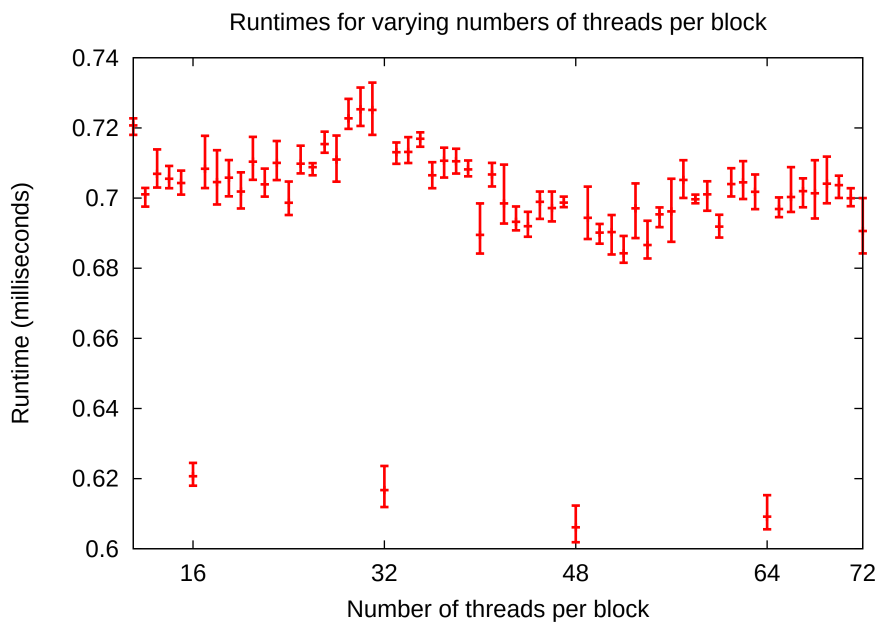


Figure 2.2: The runtime of a GPU kernel for various numbers of threads per block. When the number of threads per block is a multiple of 16, memory accesses are coalesced in this application.

time-slicing), attempting to access the Shared Memory data of another block is illegal as the order in which blocks are executed is not defined.

Texture Memory Another form of on-chip memory available is Texture Cache. Data in Global Memory can be bound to a Texture. Reads must be performed by calling a special function called a Texture Reference, and they are cached in the 6 – 8KB per SM Texture Cache. The cache is not kept coherent with changes made to the data in Global Memory, however, so data that was modified during a kernel call cannot be safely read until a subsequent kernel call. This restriction makes using Textures unsuitable for many applications, however in certain circumstances they have several advantages. Although the data is stored in Device Memory, data that is accessed as a Texture is not required to conform to the requirements of coalescing. Texture reads are also optimised for data access that exhibits 2D spatial locality. Another advantage of Textures is that they enable the usage of the CUDA array datatype, a read-only space-saving method for storing arrays. The amount of fast on-chip memory available is very limited, therefore it is advisable to use Textures to access some memory, if possible, as it is the only way of using the Texture Cache.

Constant Memory Constant Cache is a small (8KB) on-chip memory used to cache accesses to Constant Memory, which is a 64KB area of Device Memory that is declared to be ‘Constant’ (read-only). Constant Memory is optimised for data that needs to be broadcast. Reads from Constant Memory are serialised unless more than one thread in a half-warp reads from the same address, in which case only one read from that location will occur and the result will be broadcast. The access time will therefore scale linearly with the number of separate addresses that are requested. This form of memory is therefore very suitable for loading constant data onto every thread, for example runtime parameters. As was discussed in the description of Texture Memory, it is also important to use Constant Memory in order to gain access to more fast on-chip memory.

Registers Threads also have very fast read/write access to private Registers. As with Shared Memory, if each thread requires many Registers, then the number of threads that can be simultaneously active, and therefore the ability of the processors to time-slice in order to cover memory latency, will be limited. Variables that are privately declared by a thread will be automatically allocated in Registers unless they are determined to be too large or their size cannot be determined at compile time, in which case they will be allocated in Device Memory as Local Memory. Local Memory accesses will suffer the same latency as normal Global Memory and could therefore be a significant performance impediment.

2.2.4 Compute Capability

Nvidia's CUDA-enabled GPUs are available in three ranges [7]. The GeForce line is primarily aimed at the home user market, in particular for computer games acceleration. Quadro GPUs are intended for professional graphics users. For HPC GPGPU usage, the Tesla range was designed. GPUs in this range are characterised by large, high bandwidth memory, many processing cores, and lower clock speed than the other products, which reduced the heat dissipated and therefore facilitates clustering many GPUs together. Another difference of the Tesla GPUs is that they do not have graphical output, as they are intended purely for computation.

A further measure used to determine a GPU's abilities is its 'compute capability'. This is a term used by Nvidia to describe which CUDA features are available on a GPU. The original CUDA-enabled GPUs had compute capability 1.0. Higher compute capability numbers have been used to denote GPUs that have new functions available, such as atomic operations in compute capability 1.1, and increases in the number of registers per SM in compute capability 1.2. As was previously mentioned, the most important improvement was probably the introduction of double precision support in compute capability 1.3 devices.

2.2.5 Other Issues

In addition to many floating point units for performing general single-precision floating point operations, GPUs also possess special hardware to perform certain operations faster. Operations such as computing the reciprocal, sines, and logarithms are supported. It is important to note, however, that the improved performance is at the expense of accuracy.

Nvidia's GPUs are not fully IEEE-754 compliant. Some of the issues are its support of rounding modes, lack of denormalised numbers, and some operations being implemented in a non-standards compliant way.

Another issue with GPUs that may be problematic for certain applications, is their lack of error checking. Although often not present in consumer PCs, ECC (error-checking code) memory is important for applications, such as many HPC codes, where data integrity is crucial. ECC is a method by which it is possible to discover whether data has become corrupted by sources of errors such as cosmic rays. Although such errors are infrequent, if using the GPU results in more than a doubling of performance, the code could be executed twice to increase confidence in the accuracy of the data.

2.3 CUDA

Compute Unified Device Architecture (CUDA) is a programming language developed by Nvidia to facilitate the use of their GPUs for non-graphics programming (GPGPU). It is an extension of C and C++. Code written in CUDA can contain functions that run on the host, and functions that run on the device. Host functions are written using C or C++, with several additional operations available to permit interaction with the GPU. The code that executes on the device should be written in C, although certain features from C++ are permitted. There are also extensions that can be called in device code to perform GPU-specific functionality. The definitive reference sources for the language, and other technical GPU topics, are the CUDA Reference Manual [23] and the Nvidia CUDA Programming Guide [25].

The most important host extensions involve functions to allocate memory on the device, transfer memory between the host and device, and a special syntax for launching kernels on the GPU. The first and second of these have several variations, depending on the way in which the memory is allocated.

The most basic memory allocation function is `cudaMalloc`, which behaves similarly to the standard C `malloc`. It allocates the requested amount of memory on the device, and assigns the address of it to the provided pointer. If this memory is to be accessed during a kernel, then this pointer must be passed as one of the kernel arguments. The pointer can also be used in memory transfer function calls on the host.

`cudaMallocPitch` (for 2D arrays) and `cudaMalloc3D` (for 3D arrays) are slight variations that are designed to facilitate obeying the rules necessary for coalescing to be achieved. The dimensions of the array are passed, together with the pointer to which the memory will be assigned, and another variable which will store the 'pitch'. The data will be padded so that the beginning of the data block that is accessed is only required to be a multiple of sixteen elements from the beginning of the row, rather than the beginning of the entire array. As a result of this, however, accessing the data becomes more complicated and a calculation must be performed involving the 'pitch' to determine the address of elements in the array.

Another important variation of the memory allocation function is `cudaMallocArray`. This allocates memory as a CUDA array which is a proprietary format optimised for use with Textures. CUDA arrays are read-only, and may give good performance if reads exhibit spatial locality. A version for 3D arrays is also available.

The primary memory transfer function is `cudaMemcpy`, which can transfer data from the host to the device, the device to the host, and from one location on a device to another location on the same device. Several alternatives exist which are designed for memory allocated using `cudaMallocPitch`, `cudaMallocArray`, and their 3D variants. `cudaMemcpyToSymbol` allows data to be copied into Constant Memory.

Kernels are launched using the syntax:

$$\text{kernel\_name} \;<<<\; \text{GridDim, BlockDim} >>> \text{(arguments)}$$

where `GridDim` is a vector specifying the number of blocks in each dimension of the grid, and `BlockDim` is a vector containing the number of threads in each dimension of every block.

Code written in CUDA may be compiled using the `nvcc` compiler. This will split the code into parts that run on the host and parts that run on the device. The host code is compiled using a standard C/C++ compiler (`gcc` on Linux), while `nvcc` will compile the device code. The compiler has a similar interface and supports many of the same compiler options as traditional compilers. An additional option is to compile the code so that it runs in *emulation mode*, which allows the code to be executed entirely on the CPU even if there is no GPU present on the system. As CPUs do not have as many processing cores as a GPU, some serialisation of operations that would have run in parallel on the GPU is necessary. As a result, the behaviour of the code is not guaranteed to be the same when the emulation mode is used.

There are two ways to debug CUDA programs. One method is to run the code in emulation mode and use a standard CPU debugger. A problem with this approach is the code may behave differently when emulation mode is used, as discussed above. The second option is to use CUDA-GDB, which allows the debugger to work even when the code is running on a GPU. This is only possible with devices of compute capability 1.1 and higher, however.

A profiling tool is also available which provides a GUI to view a breakdown of the runtime of a GPU application. It can also display information collected from various hardware counters, such as the number of Global Memory reads and writes that were not coalesced, and the number of divergent warps (which is when a conditional statement evaluates differently for threads within the same warp, leading to serialisation). This is a very important tool as it can provide an insight into the performance of code and therefore facilitate optimisation.

Nvidia also provides libraries to implement BLAS (Basic Linear Algebra Subprograms), FFT (Fast Fourier Transform), and certain ‘parallel primitives’ (such as parallel reductions and scans). These are available as CUBLAS [22], CUFFT [24], and CUDPP [9], respectively. These were clearly designed with the aim of making them drop-in replacements for traditional CPU implementations, as the interfaces are similar (CUFFT has the same interface as the widely used FFTW, for example).

2.3.1 Example

To demonstrate how a code can be converted to CUDA and introduce some of the performance optimisation strategies that will be used in chapter 4, a simple example will be used, multiplying an N $\times$ M matrix by an M row column vector.

For execution on a CPU, this could be written in C as follows (with the array `C` previously initialised with zeros):

Listing 2.1: C matrix-vector multiplication code

```c
void matvecmult(float *A, float *B, float *C)
{

  int i, j;

  for( i = 0; i < N; i++ )
  {
    for( j = 0; j < M; j++)
    {
      C[i] += *(A + (i * M) + j) * B[j];
    }
  }
}
```

Achieving a similar operation on a GPU requires significantly more code. This is primarily because it is necessary to have additional operations that allocate and free memory on the GPU, copy data to and from the GPU, and launch the kernel. While the GPU code in this example contains more than twice as many lines as the CPU version presented, the number of additional lines should increase approximately linearly with the number of variables and therefore for more complex programs the difference may not be so dramatic.

As discussed above, GPU kernels require additional code to be executed on the CPU in order to prepare the GPU. This code must be written in C or C++ so that it can be compiled by the `nvcc` compiler. In this example it was decided to create a new function that would perform these tasks, seen in Listing 2.2.

Listing 2.2: First CUDA matrix-vector multiplication host code

```
void matvecmultgpurun(float *A, float *B, float *C)
{

  size_t pitch;
  float *A_d, *B_d, *C_d;

  /* Number of threads per block in the X, Y, and Z dimensions
   * Defined using the form 'dim3 identifier(x, y, z);' */
  dim3 BlockDim(BLOCKSIZE, 1, 1);

  /* Number of thread blocks in the X, Y, and Z dimensions.
   * Only the x dimension is being used. The division
   * N/BLOCKSIZE will be truncated to an integer. If N is not
   * a multiple of BLOCKSIZE, then one more block will be needed
   * so that all of the problem domain can be covered. This is
   * the function of the second part of the calculation */
  dim3 GridDim(N/BLOCKSIZE + (N % BLOCKSIZE == 0 ? 0 : 1), 1, 1);

  /* Allocate memory on the GPU for arrays A, B, and C */
  cudaMallocPitch((void **) &A_d, &pitch, sizeof(float)*M, N);
  cudaMalloc((void **) &B_d, sizeof(float)*M);
  cudaMalloc((void **) &C_d, sizeof(float)*N);

  /* Copy the arrays A and B from the CPU memory into
   * the GPU Memory */
  cudaMemcpy2D(A_d, pitch, A, sizeof(float)*M, sizeof(float)*M, \
      N, cudaMemcpyHostToDevice);
  cudaMemcpy(B_d, B, sizeof(float)*M, cudaMemcpyHostToDevice);

  /* Fill memory allocated for array C in the GPU memory
   * with zeros */
  cudaMemset(C_d, 0, sizeof(float)*N);

  /* Launch the kernel on the GPU with the specified
   * number of threads */
  matvecmultgpu<<<GridDim, BlockDim>>>(A_d, B_d, C_d, pitch);

  /* Copy the result array C from the GPU memory into
   * the CPU memory */
  cudaMemcpy(C, C_d, sizeof(float)*N, cudaMemcpyDeviceToHost);

  /* Free the memory allocated on the GPU */
  cudaFree(A_d);
  cudaFree(B_d);
  cudaFree(C_d);
}
```

One important feature to note is that the interface for this function is the same as that for the CPU version. This means that a function such as this could easily be dropped-in in place of an existing function that was designed to perform the operation on a CPU.

New pointers must be declared to hold the address of where memory on the GPU has

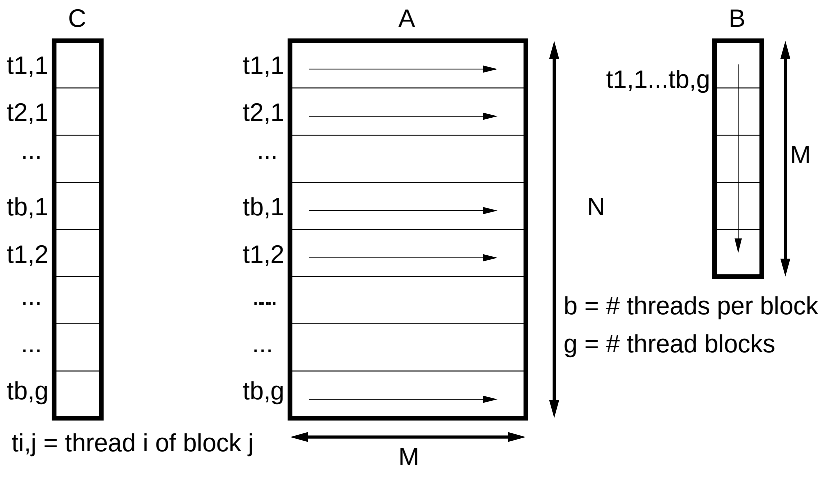


Figure 2.3: A diagram of the basic data access pattern used in the first series of GPU codes to solve the matrix-vector multiplication problem

been allocated. The convention of adding '_d' to pointers in order to denote that they point to GPU memory is useful as it is important to distinguish them from pointers to CPU memory. Such pointers can only be used by special CUDA functions.

The method used to solve this problem on a GPU is to perform all of the calculations necessary to compute one element of the result vector C on a single thread. This means that each thread will multiply a row of the matrix A by the vector B. This data access pattern is represented in Fig. 2.3. Splitting the calculation into smaller parts so that more threads could be used, for example by having each thread only multiply one element of A by one element of B, would increase the number of threads from N up to $M \times N$. Such a decomposition would introduce significant synchronisation issues as the results from all of the multiplications $A[i][j] \times B[j], j \in 1, \ldots M$, with different threads for every i and j, would need to be added together to form a single element of $C[i]$.

In this example a column of threads 'BLOCKSIZE' long is created in each thread block. If there are 'k' thread blocks, then the total of all of the threads created can be visualised as a line of threads in the x-direction k $\times$ BLOCKSIZE long. It is also specified that there should be enough thread blocks so that this line of threads is at least as long as each row of the matrix A (M). This is achieved by setting the number of thread blocks in the x-direction to be the result of dividing M by the number of threads per block, rounding the answer to the nearest integer towards zero, and adding one more block if the numbers didn't divide evenly. If the required number of threads isn't an even multiple of the number of threads per block, and so an extra block has to be used as described, then there will be too many threads created. To prevent this creating a situation where these extra threads try to access memory that is beyond the array boundaries, the code inside the kernel must be surrounded by an `if` statement that only allows threads with an ID lower than N to execute it.

Listing 2.3: First CUDA matrix-vector multiplication device code (kernel)

```
__global__ void matvecmultgpu(float *A, float *B, float *C, \
                              size_t pitch)
{

  /* Computes the global ID of the thread in the x dimension
   * by multiplying number of threads per block in x dimension
   * by ID of current block in x dimension, plus the ID within
   * the block of the thread in the x dimension */
  int i=blockIdx.x*blockDim.x + threadIdx.x;
  int j;

  if( i < N )
  {
    for( j = 0; j < M; j++)
    {

      /* As ‘A’ was allocated using cudaMallocPitch, it
       * must be accessed in this way, as instructed by
       * the CUDA Programming Guide and Reference Manual */
      C[i] += *((float*)((char*)A + i * pitch) + j) * B[j];

    }
  }

}
```

If the number of threads per block is a multiple of the warp size (32), then all of the accesses to the array C in this code will be coalesced. This is very good for performance, as it avoids the serialisation of memory access. Accesses to arrays A and B are not coalesced, however. This is because each thread in a warp will read the same column of A simultaneously, but they will be on different rows. This therefore contravenes the rule that all of the threads in the warp must access a contiguous block of memory. Accesses to B are not coalesced because all of the threads in a warp will read the same element of B at the same time. Unlike Shared Memory and Constant Memory, Global Memory does not have a broadcast mechanism, so this will result in a serialisation of access. It is possible that even with these non-coalesced memory accesses, the code could have reasonable performance because it does not use a large number of registers (only 7) or Shared Memory (48 bytes). This means that it should be possible to have many threads active concurrently, which will enable to GPU to time-slice between them to reduce the impact of memory latency. The Tesla C870 GPU on which this code was executed has 8192 registers per SM, which means with the current usage of registers over 1000 threads could be active concurrently on each SM; more than the maximum of 768 allowed. In an optimised version of this code, it would therefore be possible to use slightly more registers per thread without limiting the maximum possible number of active threads. This GPU has 16KB of Shared Memory per SM, and therefore we can afford to use significantly more Shared Memory per thread without limiting the number of active threads.

The registers used by this code are presumably storing the loop index, and perhaps the value of `pitch` and the pointer to A cast as a `char` pointer. Most of the Shared Memory used in this version of the code stores the pointers that were passed as input to the kernel function. Using the CUDA profiler it can be determined from the number of memory writes that the running sum of C is not being stored as during the loop and only written to Global Memory at the end. This means that the value of each element of C in Global Memory will be updated M times during the calculation.

The obvious first step in optimising this code for a GPU would be to create a temporary variable for each thread in the registers to store the value of C during the loop. This will use one more register, but it should reduce unnecessary accesses to Global Memory. This also eliminates the need to fill the array C with zeros before the kernel is launched as the new temporary variable that is created for each thread can simply be initialised to zero.

The next optimisation to consider would be to solve the problem of non-coalesced accesses to B by placing it into Shared Memory or Constant Memory. It can be loaded into Shared Memory with a coalesced read, or can be written to Constant Memory directly from the host. During the computation the broadcast feature of these types of memory means that when every thread in a warp reads from the same location in the array only one memory transaction will be necessary.

A complication with using Shared and Constant Memory is that they can store a very limited amount of data. Even though in this situation we would wish for the Shared Memory of each block to contain the same information – B – the available Shared Memory on each SM is in fact split between all of the blocks currently executing on the SM. To have the maximum possible number of thread blocks running concurrently on each SM, which is eight for the Tesla C870, each can use up to 2KB of Shared Memory. This corresponds to B being of length 512 (using the single precision floating point datatype). It is not possible to have exactly this number of elements in B, however, because a small amount of Shared Memory is reserved and so is not available to the user. The Constant Memory cache is only 8KB, however it is not split between the currently executing blocks, so B could have up to 2048 elements stored in the Constant cache. This would seem to indicate that Constant Memory would be the most suitable for this situation, however there is a further complicating factor to consider. To handle cases where B has more than the maximum number of elements that can fit into one of these memory areas, it is necessary to split the calculation into stages. This would involve performing the operation of adding to $C[i]$ the results of multiplying $A[i][j]$ by $B[j]$ for *j* from 0 to $k-1$, where k is the maximum number of elements that can be stored. Once this has finished, the stored values of B are replaced with elements k to $2k-1$, the calculation repeated for these values of *j*, and so on until all elements of B have been completed. With Shared Memory this could be accomplished quite easily by having a `__syncthreads()` barrier to synchronise all of the threads in a block after loading the new data, to make sure all of the new data has been loaded, performing the calculations of the current stage, and synchronising again to ensure all calculations have completed before the data for the next stage is loaded. To achieve a similar effect with

Constant Memory, the maximum amount of data, 64KB, would be loaded into Constant Memory from the host before the kernel was launched. The kernel would only perform the calculations for the values of B loaded and then finish. Once this was completed, the host would load the next section of B into Constant Memory and then launch another kernel to perform the next stage of calculations, and so on. It is likely that this would incur a significant performance penalty.

Both of the proposed solutions have problems. In the Shared Memory approach, two thread block barriers per 2KB of B would need to be called. Nvidia claims that these barriers are fast, however they would of course still cause a delay. Every thread block would have to separately load the current segment of B, even though they will all be loading the same data. This is inefficient, but is unfortunately the only way it can be achieved. In the proposed Constant Memory method, many kernel launches would be necessary if B was large. The contents of the on-chip memory – registers, Shared Memory (which will store the input pointers even if it is not used for anything else), and the Texture and Constant caches – are lost between stages in this situation as they are not persistent between kernel launches. The most serious of these is the registers data because it will contain the values the array C contained at the end of the stage: the result of all the operations performed so far. This means that in the Constant Memory version of the code, the array C would need to be written to Global Memory at the end of each stage and then read again at the beginning of the next. This would not need to be done in the Shared Memory version because each thread block remains active until it has completed doing the computation for all of B and so the only Global Memory operation involving the array C would be writing its final value. These kernel launches in the Constant Memory version would only need to occur for every 64KB of B, and so the reduced frequency of new stages compared to the Shared Memory version might compensate for the delay. Another potential issue with Constant Memory is that the programmer can only write data into the Constant Memory space in Device Memory; transferring this into the Constant cache is controlled automatically by the GPU and not by the programmer. This means that performance is dependant on how sophisticated the mechanism that controls the Constant cache is.

The argument for having as many thread blocks concurrently active as possible was already presented: maximising the SMs' ability to time-slice. It was also discussed, however, that having more thread blocks active means that the amount of Shared Memory available to each is reduced. These two issues are therefore in contention. Determining whether best performance can be obtained by having as many thread blocks as possible active, or by loading a larger segment of B into Shared Memory to reduce the number of stages, can only really be done through experimentation.

Using Shared Memory without employing the multi-stage approach for this modification results in the amount of Shared Memory being used increasing by the size of the array B, while the number of registers used decreases to 5. It is not clear why the number of registers should decrease, but it may be due to changes in the way that the code is compiled, for example differences in how much the loop is unrolled. Without using stages, this code can only be used when B contains less than 500 values. If the multi-stage approach that was described is used, then the number of registers per thread increases to 11 and the code becomes considerably more complicated. Having 11 registers per thread is a problem as it restricts the total number of threads that can be active concurrently to less than the allowable maximum. It is possible to force the compiler to limit the number of registers per thread. This will be done by either not using a register where the compiler thinks there should be one, or by using Local Memory to store the excess register values. The former of these would result in more computations to recalculate what would otherwise have been stored. It is possible that the performance improvement from the increase in the number of active threads may outweigh this additional overhead. The latter possibility, however, may damage performance as accesses to the Local Memory registers would take at least 200 cycles compared to one cycle for when the data was stored in the Registers.

The Constant Memory implementation of the code (version 5) only uses 6 registers per thread, and therefore the maximum number of thread per block can be used.

A summary of the versions of the code described so far is available in Series 1 of Table 2.1. The performance of these codes compared to the CPU implementation of Listing 2.1 can be seen in Fig. 2.4. Note that the timing results used to generate this graph are only for the computation kernel; they do not include the time associated with allocating, freeing, and transferring memory to and from the device. Each datapoint was obtained by executing the appropriate version of the code for various numbers of threads per block, and then the fastest of these results (i.e. the number of threads per block that resulted in the best performance) was used.

The advantage of using the on-chip memory – Shared and Constant cache – can be clearly seen in Fig. 2.4. For the configuration M=64, N=4096, for example, version 4 of the code (which uses Shared Memory) is over six times faster than version 1 (which only uses Global Memory). The advantage of having larger numbers of threads can also be clearly seen. When a problem can only be decomposed into a small number of threads, there may not be enough thread blocks for every SM to be occupied. This leaves SMs idle and so the GPU's computational power is not being fully utilised. Increasing the number of threads further will often continue to improve the performance. As the number of rows in matrix A (N) increases, the number of thread blocks containing at least one warp of threads that the computation can be decomposed into increases. This ensures that there is enough work for the SMs to effectively use their time-splitting abilities to cover memory access latencies. This can be seen by the rapid increase in performance of every version of the code as N increases.

Eliminating the unnecessary writes to C in Global Memory between versions 1 and 2 did not appear to have a noticeable effect on performance. It is likely that the non-coalesced reads of A and B every iteration took far longer than the coalesced writes to C, and so the cost of the writes was hidden.

Versions 3, 4, and 4mr of the code all have almost identical performance. This is probably due to the same reason that version 2 was not faster than version 1: version 4 eliminates the non-coalesced reads of B that occur in version 3, but reads of A occurring at the same time are non-coalesced, so the total memory access time (which will be determined by the slow non-coalesced reads of A) is unchanged. It is slightly surprising, however, that there is no noticeable different in the performance when the number of registers is restricted to 10 in version 4mr. It was expected that, as doing this would allow more threads to be active simultaneously, memory access latencies would be partially hidden. The likely explanation for this lack of improvement is that there is not enough computation per thread between memory accesses, so even when more threads are active, the SMs still spend most of their time idle while waiting for data from memory.

It appears that the Constant Memory implementation of the code (version 5) is inferior to the Shared Memory one (version 4). For M greater than 128, their performance is identical, while for smaller M the Constant Memory version is slower. This could be attributable to any (or a combination of) the possible causes for this described earlier: the overhead of relaunching kernels, additional Global Memory accesses due to lost on-chip memory between kernel launches, and relying on the Constant Memory cache to perform well.

One of the most striking features of Fig. 2.4 is that for increasing M, the performance of every GPU version relative to the CPU version initially increases, and then begins to decreases. This is caused by the non-coalesced accesses to A. As M increases, so does the amount of work for each thread allowing the GPU, which was underutilised for smaller M, to increase its speed-up over the CPU version. The increasing parallelism obtained by having more threads (as N increases) can produce great performance improvements at this stage without restriction. This GPU can have up to 6144 threads active simultaneously, which means that even for the largest N tested, the GPU is still not being fully utilised. It is likely that the performance at these small values of M could be improved further if larger N was used. Between $M = 64$ and $M = 128$ this regime is seen to change dramatically. Every time a thread reads an element of A, a half-warp of elements (16) must be read. This means that during the calculation, the entire A matrix will need to be loaded from Global Memory 16 times. With increasing M, the number of times each thread must read from A also increases. It is believed that at this point reading from Global Memory becomes a bottle neck. For these higher values of M, increasing N still initially results in a performance improvement due to increased parallelism, however for high N the performance actually decreases. For small N, the limit of Global Memory bandwidth has not yet been reached, so the performance is unaffected by increasing M, as can be seen in the graph. For large N, however, the Global Memory bottleneck appears because of the larger number of threads attempting to simultaneously access it. This effect begins to present itself for smaller N as M continues to be increased. This is what would be the expected case based on the explanation postulated. The peak Global Memory bandwidth of the Tesla C870 GPU that was used is 76.8GB/s. For $M = 128$, $N = 4096$, loading A alone from Global Memory (with non-coalesced reads) if peak transfer speed was obtained would take 4×10^{-4} seconds. Version 4 executes in 9×10^{-4} seconds. This clearly supports the memory-bound explanation. Versions 1 and 2, which have a longer runtime and therefore the memory

Version	Description
Series 1	
1	Original version presented in Listing 2.3
2	version 1 with one register per thread used to store intermediate value of C
3	version 2 with all of B stored in Shared Memory
4	version 3 with modification that B is loaded in stages to avoid using too much Shared Memory
4mr	version 4 compiled with instruction to compiler that only 10 registers should be used
5	version 2 with B loaded in stages to Constant Memory
Series 2: same as Series 1 but using the transpose of A	
6	version 1 with modification to use A transpose
7	version 2 with modification to use A transpose
8	version 3 with modification to use A transpose
9	version 4 with modification to use A transpose
9mr	version 4mr with modification to use A transpose
10	version 5 with modification to use A transpose

Table 2.1: Summary of the GPU code versions used in the matrix-vector multiplication example

bottleneck would have a less pronounced effect, appear to be less affected than the other versions for increasing M until $M = 1024$ (not visible in the data presented), at which point all versions have approximately the same runtime. These two version do have a higher number of non-coalesced Global Memory accesses, however, since reads of B are not coalesced. Reads to B are particularly sensitive to increasing N because reads to it are not just non-coalesced, but also clash for every active thread since they will all try to access the same element of B and Global Memory does not have a broadcast mechanism. This is probably the reason why the performance degradation at very high N in versions 1 and 2 appears to be more pronounced than in the other versions.

The non-coalesced accesses to A are obviously the next element of this code that must be addressed. This example is very illustrative as this next stage clearly conveys the difficulty of obtaining excellent performance using a GPU. The other optimisations previously described did not require more than a basic understanding of the code. In order to transform accesses to A into coalesced reads, it is necessary to comprehend the structure of A and the data access pattern. In the case of matrix-vector multiplication, this is not very difficult, but such insight into the functioning of large, complex codes is usually only possessed by the code author. For this reason, obtaining dramatic performance improvements by implementing a code one is not very familiar with is unlikely.

The key step in understanding how to make accesses to A coalesced is to realise that consecutive threads will read from neighbouring rows of the same column, and will

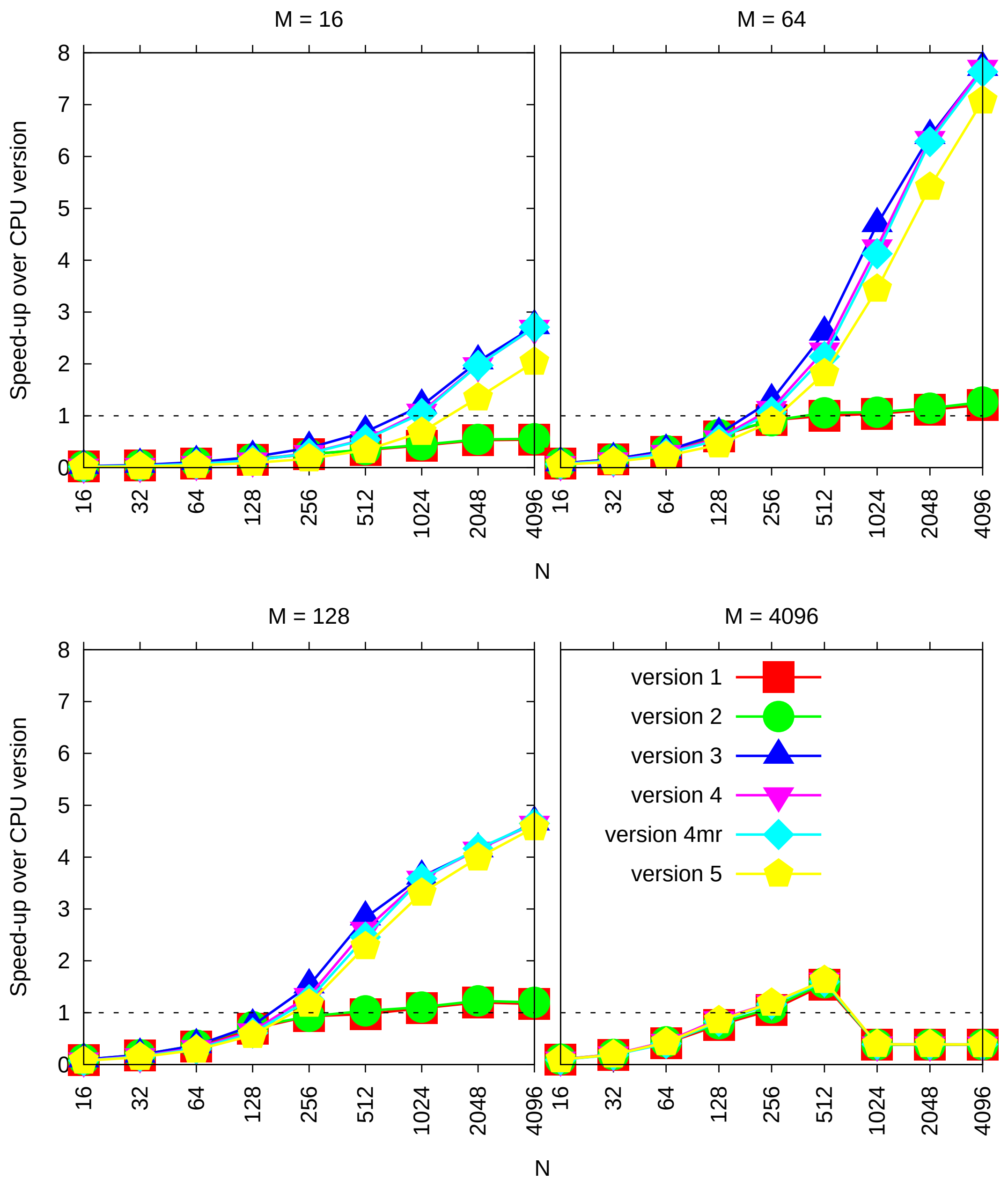


Figure 2.4: Speed-up of GPU versions compared to CPU implementation of Matrix-vector multiplication for $N \times M$ matrices, where M and N range from 16 to 4096. It was not possible to run version 3 for M=4096 as each thread block would have used more Shared Memory than is available per SM.

sweep across the columns in this formation. For coalesced access, threads in a half-warp must access a contiguous block of memory, which is not the case in the current situation. If the matrix A is transposed, however, then neighbouring threads will read from neighbouring columns of the same row for every loop iteration. This will therefore qualify as a coalesced access.

Performing this modification only requires that a transposition function be created and that the indices in the access to the array A in the kernel be swapped. This small change results in a very large performance improvement, however, as can be seen in Fig. 2.5. As in the results for the first series of tests (without using the transpose of A), the speed-up over the CPU version increases quickly as the work per thread rises for larger M. As reads to A are now coalesced, the amount of memory that must be transferred from Global Memory to load A is decreased by 16 times. The explanation that was presented for the decreasing performance after $M = 64$ in Fig. 2.4 – that the limit of Global Memory bandwidth had been reached – appears to be verified again as a noticeable drop in performance relative to the CPU version is now observed around $M = 1024$. At $M = 1024$, the matrix A contains 16 times the number of elements it did when $M = 64$. As the accesses are now coalesced, the same amount of data will need to be transferred from Global Memory in both cases.

A remarkable feature of the series 2 graph (Fig. 2.5) is the excellent performance of version 8. It significantly out-performs the other versions over the range of array sizes for which it can be used ($M < 4084$). This suggests that for this problem, storing as much of B as possible in Shared Memory is more important than having the maximum number of thread blocks concurrently active (version 9). Another probable reason for version 8's better performance than version 9 is its reduced use of registers. As was also visible in series 1, limiting the number of registers used per thread to avoid limiting the number of active threads in this way does not improve performance for this code. This indicates that the steps the compiler must take to constrain the number of registers in this way, such as storing register values in Local Memory, are reducing performance. Another notable feature of the results is that the Constant Memory implementation (version 10) does not out-perform the Shared Memory code in any case.

It is important to understand that although very impressive improvements in the performance of calculating the result of the matrix-vector multiplication were achieved, the data presented does not include other runtime costs that must be borne in order to use this fast GPU implementation. These costs are primarily associated with preparing the memory of the GPU for the calculation kernel to take place. Memory must be allocated on the GPU to store the arrays A, B, and C. A and B have to be copied from the CPU memory into the GPU memory and the result vector C must be copied back after the calculation. Finally, the memory on the GPU must be freed. As is evident from Fig. 2.6, the cost of these operations can be very significant.

When all of the additional costs are included, performing a single matrix-vector multiplication on the GPU is only faster than the CPU version for the top performing cases, and even then it is only slightly faster. This code would therefore only be useful if it were employed in an application where it was called many times with the same A, but

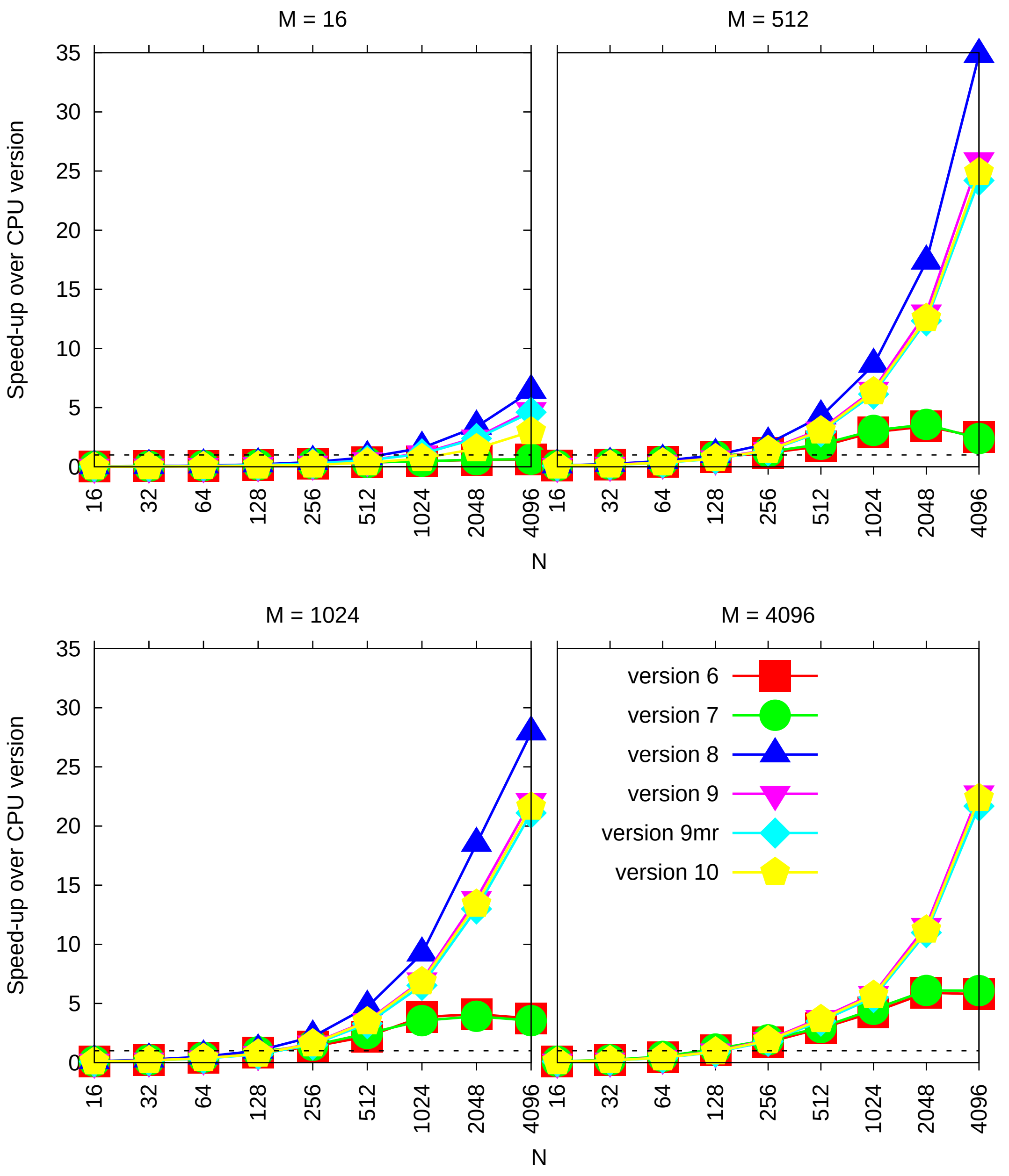


Figure 2.5: Speed-up of GPU versions (using the transpose of the matrix) compared to CPU implementation of Matrix-vector multiplication for $N \times M$ matrices, where M and N range from 16 to 4096. It was not possible to run version 8 for $M = 4096$ as each thread block would have used more Shared Memory than is available per SM.

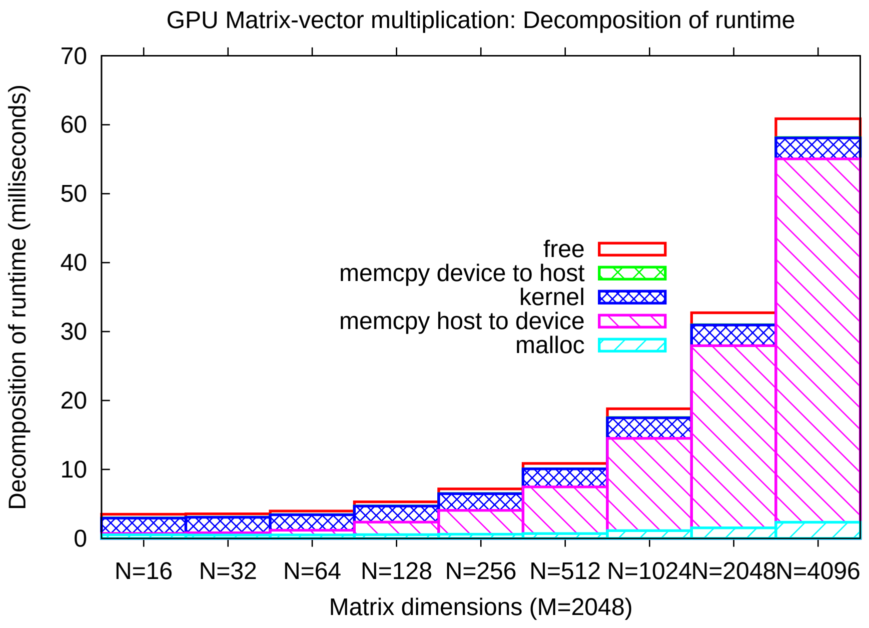


Figure 2.6: Breakdown of runtime for version 9 of the code in the case where $M = 2048$. The workload per thread is independent of N, so the kernel time does not change. Note that the time taken to transpose A is not included.

varying B. In this situation the allocation and freeing of memory on the GPU would only need to be done once each. The largest benefit, however, would come from avoiding the need to transfer A repeatedly. The speed-up obtained would then be close to that presented in the earlier charts.

Thus one of the most important steps in porting a code to run on a GPU is to minimise the amount of data that must be transferred between the host and device. This would be done principally by examining the code to determine which data is modified between kernel launches and only transferring that data.

2.3.2 Alternatives to CUDA

Several other languages were designed for programming GPUs, such as BrookGPU [19], ATI Stream SDK (formerly CTM) [1], and the RapidMind Development Platform [31]. Currently usage of these languages is small compared to CUDA.

Other methods for utilising GPU resources are based on CUDA but present a different interface to the developer. The Portland Group, a compiler provider, has developed compilers that automatically convert compiler directive specified sections of C or Fortran code to CUDA [28]. The Portland Group is also developing a Fortran CUDA compiler that should facilitate the porting of Fortran codes to use GPUs [29]. Another project with similarities is Flagon [12], which is a wrapper library that allows CUBLAS functions to be called from Fortran 9x. CUDA-Lite [37] is an attempt to create a tool that would automatically modify code to employ the advanced functionality of GPUs, guided by the programmer. CUDA-Lite is not currently openly available, however.

Another alternative will be provided by OpenCL [27]. This is an open standard for parallel programming, managed by the Khronos Group, and supported by many manufacturers including Nvidia. The particular emphasis of OpenCL is the support of heterogeneous architectures. It is envisaged that it will be possible to run code written using the OpenCL environment on a mix of CPUs, GPUs, and Cell processors. The standard is still in development, but with the continuing rapid progression of CUDA, it is unclear whether it will be widely adopted for GPU programming.

2.4 Applications

In this section, the applications that were considered for GPU acceleration will be introduced. This includes codes that it was ultimately decided were unsuitable for porting to a GPU.

Ludwig was chosen as one of the codes to be considered because Dr. Alan Gray, one of the supervisors of this dissertation, was familiar with it and believed that it may be suitable for GPU acceleration. The other codes were selected because people involved with them expressed interest in having the suitability of the codes for GPU execution investigated.

2.4.1 CENTORI Optimisation Library

CENTORI is a code developed by the United Kingdom Atomic Energy Authority. It is designed to model plasma turbulence in a tokamak using a fluid model. This is important for fusion energy research, such as the ITER project. The code is not publicly available, and so little information has been published that describes it. One source, however, is [17], which briefly describes the code and its purpose, and contrasts it with another code that has a similar purpose but employs a different method.

CENTORI must perform many mathematical operations to accomplish its goal. Optimising the execution of these operations for a variety of architectures is the subject of a PhD project at the University of Edinburgh. To facilitate this, the mathematical operations have been isolated and generalised so that resultant library of functions comprises simple and familiar operations such as multiplications and additions involving matrices and scalars. The simplest code to achieve each of the desired operations is included, referred to as the ‘naive implementation’ or ‘canonical example’, and used for comparison with more sophisticated versions that are optimised for the target architectures. Some performance data for this library was presented in a technical report comparing HPCx and HECToR [14].

The operations that were provided for GPU acceleration consideration included: matrix addition, multiplying a matrix by a scalar, the gradient operator, the divergence operator, the curl operator, the dot product, the cross product, computing the square of the $\ell 2$ norm, and matrix-matrix multiplication. In total there were 13 operations.

2.4.2 CHILD

The Channel-Hillslope Integrated Landscape Development (CHILD) Model is a code originally developed at MIT, but now contributed to by researchers at many institutions. It seeks to simulate geomorphic processes such as erosion, and hydrological processes such as changes to catchment areas. An overview of the techniques used in the code is presented in [36] and [35]. A document that discusses the code implementation, and references to journal articles documenting research that involved the use of CHILD, are available on the application's website [4].

CHILD is a serial code written in C++. An unstructured mesh is used to model the landscape. The reasons why this method was chosen rather than a regular grid are discussed in [36]. The path through the code followed when the program executes is determined by the input used, with different functions called depending on what the input file aims to model.

2.4.3 CSMP

The Complex Systems Modelling Platform (CSMP) is designed to facilitate simulating physical processes, particularly those involving complex geometries such as hydrological applications. The code employs several numerical methods, such as combined finite element-finite volume, finite-difference, and particle methods. It uses double precision arithmetic so that materials whose properties vary over orders of magnitude can be reliably handled. CSMP is a parallel code written in C++. It is primarily developed at ETH Zurich and Imperial College London. The code's website, [6], contains extensive documentation such as [20] which describes the design and functioning of the code.

CSMP is designed to work with external software to perform tasks such as designing meshes. One important external package is SAMG. This is a commercial library for solving systems of equations. CSMP does, however, contain a free implementation of this functionality, so it is still possible to use it without SAMG.

2.4.4 FAMOUS

FAst Met Office/UK Universities Simulator (FAMOUS) is a climate modelling application developed by a consortium including the Hadley Centre at the UK Met Office. It is a lower resolution version of another climate model called HadCM3. Being of lower resolution allows it to simulate longer timescales; [33] states that it can process about 100 years in one day on an 8-node cluster. Further details are available in [16].

FAMOUS is written in Fortran 77 and contains over 700 sources files. The code uses double precision, however a single precision version is available. One high-profile use of FAMOUS was for the Millennium experiment on `climateprediction.net`, which aimed to simulate the climate from 800AD to the present day.

2.4.5 Ludwig

Ludwig is a Lattice-Boltzmann application used to simulate binary fluids. A detailed description is available in [10]. A regular 3D grid is used to decompose the problem domain, and due to the load-balanced nature of the major elements of the code, this is evenly distributed among the available processors. The key data in the application is the velocity distribution of the two fluids, which is stored in the `site` structure for each grid element. The code is highly parallelisable as some of the time-consuming functions performed every iteration, such as the collision computation, do not involve inter-grid element communication.

The code is written in ANSI C and can run in serial, or use MPI for parallel execution. It is developed by EPCC at the University of Edinburgh.

2.5 Related work

Many successful attempts to accelerate applications using GPUs have been reported. The Nvidia CUDA website has a showcase of submitted GPGPU acceleration results [8]. Because GPGPU has only recently become widely accessible, primarily since the introduction of the CUDA language, only a few such cases have been documented in scientific journals.

In an earlier project, I implemented a code to reverse a simple edge-detection algorithm on a GPU [32]. A speed-up of up to 7 times over a CPU version of the code was obtained. The performance was limited by lack of global collective operations, such as a global barrier, or some form of message-passing facility. This meant that the only way to ensure threads in neighbouring blocks had completed the current iteration so that a halo-swap could be performed, was to launch a new kernel for each iteration. As each iteration contained very little computation, the overhead of repeated kernel launches was a significant element of the runtime.

A group in Germany have used GPUs for the acceleration of a Monte Carlo simulation of the 2D and 3D Ising model [30]. Using an Nvidia Geforce GTX 200 GPU, the authors managed to obtain a speed-up of up to 60 times for the 2D model and up to 35 times for the 3D model, compared to a single core of an Intel Core 2 Quad CPU. This GPU is a high-end model in the consumer graphics-targeted Geforce range, with

240 1.3 Ghz cores. An earlier attempt by another group made in 2005, before the introduction of CUDA, only obtained a 3 times speed-up [34]. The authors of the more recent paper implemented a random number generator on the GPU. This meant that only the seeds for the generator needed to be transferred to the GPU, reducing the memory transfer overhead. A checkerboard algorithm was used so that updates could be made independently on the distributed array of sites, however each iteration still required the results of other threads' calculations from the previous iteration, and so a new kernel had to be launched for every iteration. Shared memory was used for the seed values as they were required several times by threads, and was also used for a binary tree structure that was needed to compute the reduction of the values within a block. The paper does not discuss efforts made to ensure that memory accesses were coalesced. The checkerboard algorithm would have made coalescing difficult, due to the requirement that threads in a half-warp access contiguous chunks of memory.

Using GPUs for Doppler exoplanet searches resulted in a speed-up of up to 600 times [13]. It is important to note, however, that this speed-up is based on comparing a single precision GPU code with a double precision CPU version. The largest performance improvements that this group from the University of Florida achieved were due to their understanding of the code and science, rather than GPU expertise. The first of these was identifying which parts of the code required double precision floating point data. Converting the rest of the code to single precision resulted in the GPU performance increasing by about 11 times. Another major improvement was made by realising that by doing an additional calculation on the GPU, the amount of data that needed to be transferred back to the host could be greatly reduced. The memory transfer bottleneck between the host and device means that such a reduction can dramatically reduce overall runtime. Attempting to optimise the code by using GPU features such as Shared, Constant, and Texture Memory only increased the performance by a few percent.

A single-precision, high-order finite-element earthquake model was ported to a GPU, obtaining a speed-up of up to 25 times [18]. The key to achieving this was implementing a 'colouring scheme' to prevent Global Memory write-conflicts by different thread warps. This is a method of decomposing a connected mesh into disjoint elements. Further optimisation was obtained through the use of GPUs' on-chip memory, and by restricting the number of registers per thread so that the number of threads per SM was not limited by register usage.

An understanding of the Physics was again used to modify a code so that it was more suitable for GPU execution in [11]. This involved arranging the data for a Lattice QCD code in a form that efficiently used the memory layout on a GPU. The article describes this for the situation before CUDA was introduced, when code had to be written in languages designed for graphics processing, and so memory had to be stored in the form of pixels. Although the interface presented to the programmer no longer reveals this, the hardware is still optimised for such a data layout and so the same approach was used more recently in a CUDA Lattice QCD code [2].

As stated in section 2.4, Ludwig (one of the applications considered for GPU acceleration in this dissertation) uses a Lattice Boltzmann method. Porting Lattice Boltzmann codes to run on GPUs has been performed previously. One such example is [39], where speed-ups ranging from 25 to 184 times were claimed. Another more recent attempt, described in [38], obtained speed-ups of up to 38 times.

Chapter 3

Porting

Porting – modifying a code so that it runs on a GPU – is the first step towards GPGPU acceleration of an application. Because of the substantial differences between CPU and GPU architectures described in chapter 2, further optimisation work is usually necessary to enable the code to fully utilise the GPU's capabilities. Good performance is therefore not expected after porting alone.

The process of porting a code can be divided into four distinct stages: determining whether a code is suitable for GPU execution, and if it is, which section of the code will be ported; locating all of the data that will be required by the code on the GPU, and ensuring that it is transferred to the GPU; converting the code to C (if necessary); and decomposing the GPU code into many threads. Another important element of writing code for a GPU is debugging, which will also be discussed.

3.1 Selecting code to port

Converting a large and complex HPC application to employ GPU acceleration is a difficult task. The first stage of porting an existing code to run on a GPU is to identify which functions in the code will be targeted for acceleration. As has been previously indicated, the architecture of GPUs means that only certain types of codes are suitable for execution on a GPU. The code must be examined to determine whether it possesses the necessary characteristics. HPC codes are frequently many thousands of lines long, however, so an efficient examination should be guided by the result of profiling the program. A code profile will show which functions consume the most runtime, and therefore, as a result of Amdahl's law, would potentially produce the greatest overall performance improvement if accelerated. The function or section of code identified should ideally be embarrassingly parallel, which means that it can be executed independently by many threads. As code that is executed on the GPU must be written in C, it is obviously necessary that the code is already written in C or can be rewritten so that it is. If the GPU portion of the code needs to be converted to C, then there needs to be a way to interface between the C and CUDA code compiled using `nvcc` and the rest of the code. Code on the GPU cannot access functions that execute on the CPU, so any such functions must also be converted to run on the GPU. GPUs have quite a large amount of memory, especially high-end models, however they contain less than most CPU-based HPC systems. Another check to make, therefore, is that the memory that needs to be accessed is within what is available. This issue could often be resolved by launching series of kernels in turn, each operating on a different part of the data. While it should be possible to make code that conforms to these conditions run on a GPU, to have a chance of obtaining good performance it is also desirable for the code to have high numerical intensity, and preferably to operate in single precision.

Ludwig A profile of Ludwig using `gprof` showed that about 80% of the code's run-time was being spent in the collision function (see Appendix C.1). Examining this part of the code showed that it was suitable for GPU execution as it consists of a loop over the elements in the 3D problem domain with no interaction between the calculations for different elements. The function also contains quite a lot of floating point operations per element, which indicates that it may be possible to obtain a good speed-up with a GPU.

FAMOUS No function in FAMOUS consumed more than about 10% of the total runtime (see Appendix C.2). Porting a single function would therefore not produce a significant improvement in the application's performance. Many of the functions examined operated on the same data, so if a large portion of the code was executed on a GPU then this data would only need to be transferred onto the GPU once, avoiding this bottleneck. Due to time constraints it was not possible to port more than one function to run on a GPU. All of the functions that consume the most runtime operate similarly. As there was no distinguishing characteristic to suggest which of the functions would be the most appropriate to port, it was decided to port the function that was responsible for the most runtime.

CENTORI In the library of optimised functions for CENTORI, the task of isolating simple compute-intensive kernels had already been performed. It was decided to port all of the kernels in the library so that the performance of the GPU could be investigated for many different types of operations. It was anticipated that some of the kernels would probably not benefit from GPU acceleration because they did not contain enough floating point operations per data element loaded. Despite this, the overall performance of CENTORI would probably benefit from running these kernels on the GPU as the need to transfer data between the CPU and GPU would reduced if, for example, one of these operations occurs between two other operations performed on the GPU. Despite some of the kernels lacking compute-intensity, all of the kernels do satisfy other desirable properties for GPU execution, such as performing the same operation on every element of arrays independent of the results of other elements. As the library was designed to test novel methods of optimising the operations it performed, it already contained performance measurement and verification features, eliminating the need for these to be developed.

CHILD As can be seen in the profile of CHILD in Appendix C.3, the functions responsible for the most runtime only do so because they are called a very large number of times during the code execution. GPUs perform best at computationally intensive tasks, so these functions are not appropriate. CHILD is also written in C++ and uses the object-oriented features of this language. Code that executes on the GPU must be written in C, and so it is likely that large changes to the structure of the code may be necessary in order to port it. Another problem with CHILD is that it uses an unstructured mesh, with linked lists being used extensively to navigate between elements. Achieving good GPU performance with such a data layout would be challenging as it would be unlikely that memory access coalescing would be possible as neighbouring threads would probably not access contiguous blocks of memory. Furthermore, communication with one of the lead developers of the code revealed that he did not think the code was suitable for any form of parallelisation. It was therefore decided not to continue with work on this code.

CSMP The CSMP code exhibits some of the characteristics desirable for GPU acceleration, such as operating on large arrays, and having an existing parallel implementation (suggesting that the code is suitable for parallel execution). It is, however, written in C++ which may present some difficulties. As was discussed in section 2.4, CSMP requires access to an external linear equation solver. A license for the proprietary SAMG package that is usually used for this with CSMP was not available, and so a free alternative was used. Unfortunately this made compiling and using the application problematic, which resulted in there being insufficient time to conduct the profiling and examination of the code necessary to determine whether it was suitable. No further work was therefore performed on this code.

3.2 Identifying data and code required by GPU

Once a suitable section of code has been selected, the next task is to identify all of the data that must be transferred to the GPU in order for it to execute the code. How difficult this is to accomplish depends heavily on the design of the original code. If the chosen code is a single function that does not call any other functions, and only accesses data that is passed to it (by reference or value), then this step is trivial. On the other hand, if the code accesses many variables and complex data types with global scope, or calls functions in other files which in turn access variables that have file scope, then this process could be quite time-consuming. As having lots of inputs to functions is often discouraged, especially when several functions have similar inputs, global variables are common. It is also frequently considered to be good practice to have many small functions rather than several large functions. This means that it is often the more difficult of the scenarios presented that will be encountered.

At least two functions will need to be created or modified in order to use the GPU. The first, which I will call the GPU host code, runs on the CPU and has the purpose of declaring pointers to GPU memory, allocating memory on the GPU for these pointers, copying data from the CPU memory to the GPU memory pointers, launching the GPU kernel, copying results back, and freeing the GPU memory. The second will be the GPU implementation of the function that was selected for GPU execution. Additional functions may be required for GPU implementations of any other functions that are called by the target function. For all of the codes ported it was decided to create a new file for the GPU code, rather than modifying functions in their original files. This was done to avoid interfering with the rest of the host code, for example functions called by the GPU code may also be called by the host code. It is also a decision that will usually be forced on the developer as the GPU code must be compiled by the `nvcc` compiler. This may not be suitable or possible for the existing host code files, which will often contain other non-GPU functions, and may not even be written in C/C++. It is also currently a requirement of the `nvcc` compiler that all GPU functions are located in the same file as the GPU code that calls them, which would not normally be the case in a large, multi-file code. Doing this, however, means that data that may have been available through file scope identifiers, is no longer accessible. For example the function being ported may call a function in another file that uses file scope identifiers to perform its operation. Giving these identifiers global scope and declaring them to be external in the GPU host code file should make it possible to access them so that they can be copied. In more complicated situations, such as if several files have file scope identifiers with the same name, then changes may need to be made to the host code (for example re-naming the variable).

As GPUs are optimised for single precision computation, it would be preferable if most operations in the GPU code could be performed at this level of precision. When this is not possible, and a suitable GPU (compute capability of at least 1.3) is available, then double precision can be used. If this is going to be done, then it is very important to compile the GPU code with the `-arch=sm_13` flag to inform the CUDA compiler of this. Without the flag, the compiler will silently convert all double precision operations on the GPU into single precision. This does not mean that the code can simply be used without the flag to run the code at lower precision. The data on the GPU will still be in double precision, as it will have been allocated as such in the GPU host code and double precision data will have been copied onto it from the host. When this data is used in single precision floating point operations without first converting it to single precision (which the GPU will not do), the result will be garbage. It is not currently possible to enable warning messages on the compiler that would inform you when this automatic conversion to single precision is being performed.

Ludwig Ludwig falls into the more difficult category described above. The collision function that was ported does not have any inputs passed to it. It was therefore necessary to carefully scrutinise the code to find all of the data that it accesses. The program `cscope` was vital for this. This is a utility that creates a map of C source code files and provides an interface to several facilities for using the map, such as finding where identifiers were declared or accessed. Another utility called 'Exuberant `ctags`' performs a similar function, and supports many programming languages. With such tools it was possible to quickly identify where identifiers were originally declared, so the datatype of identifiers and the size of arrays could be determined. These two pieces of information are essential as they are necessary to correctly make a copy of the data on the GPU. Pointers of the correct datatype are needed to store where the data will be located on the GPU. The array sizes need to be known so that the correct amount of space can be allocated in the GPU memory and so that the data can be copied to the GPU. In Ludwig many of the arrays were dynamically allocated in the CPU's memory, with the size being determined by calculations involving local or file scope identifiers. In order to allocate the correct amount of data on the GPU it was necessary to make the result of these calculation available in the GPU host code. This could either be done by making the identifiers used in the calculation available so that it can be re-evaluated in the GPU host code, or just the result of the calculation. Neither option is attractive as they may both require modifying the original code, possibly introducing errors. It was decided to use the latter option as it was desirable to avoid making file scope-only identifiers global when possible. There were quite a large number of file scope identifiers (declared at file scope with the `static` modifier) for which it was necessary to give global scope so that they could be accessed by the GPU host code. This approach was used because, unlike the size of dynamically allocated arrays, the data of the identifier is likely to change during the program execution, so making a copy of it might result in the GPU not receiving the correct data. The code contains a number of structures. These were slightly more complicated to port as it was necessary to ensure that the structure declaration was also available to the GPU host code in addition to the declaration of the identifier that is in the form of the structure. There are several scalar values that are used in a few of the functions called by the main collision function. To avoid having to pass pointers to these values into the collision function when it is being called from the host and then pass them again from the collision function to the functions that use them, it was decided to have them as file scope GPU variables. As the GPU code file was becoming quite long it was desirable to split it into multiple files. It was envisaged that there would be one file containing the GPU host code, another containing the GPU code (as previously discussed, all GPU functions must be in the same file), and a header file, which would contain function declarations, and the declarations of the file scope GPU variables just mentioned. It was necessary for these variables to be declared in the header file as they would need to be accessible to the GPU host code so that data from the CPU memory could be copied into them, and from the GPU code so that they could be accessed from the GPU. Unfortunately it was discovered that the CUDA compiler does not support linking GPU variables from external files in this way. The only solution was to combine all three files into a single file.

FAMOUS Locating all of the data that needed to be transferred to the GPU was very simple for FAMOUS. This was because all of the identifers accessed were passed to the function as inputs. As well as eliminating the need to search the code in order to find out which data the code accesses, this also had the advantage of making it easy to find out the datatypes and the sizes of the arrays, as this information was presented at the start of the function's code. This function also had noticeably fewer variables than the function ported in Ludwig, and most of the arrays were of the same size (which was not the case in Ludwig), so the task of allocating, and copying data to and from the GPU required considerably less effort. This function (and many of the other functions examined) did not call any other functions, so it was not necessary to port any function other than the one chosen. It was a concern that calling a function written in C and compiled using `nvcc` from FAMOUS, which is written in Fortran 77, may present a difficulty. This worked very well, however. The C function must have an underscore appended to its name, and the CUDA runtime library must be included when the CUDA code is being linked with the Fortran. One small difficulty that did arise, however, was that the function declared an array of size that could only be determined at runtime. Variables declared locally within a function are usually allocated in registers when porting to the GPU, as they are normally for temporary storage. It is not possible to declare an array in registers, whose size is not known at compile time. Investigating this array further showed that it was of the same size as most of the other arrays, and contained temporary storage for every element. This was necessary due to the structure of the code, which contained several loops over all of the elements, rather than one large loop, which would have permitted having this variable store data for the single element that was being processed in the current iteration. As the GPU version of the function would assign one thread to each loop iteration, this variable no longer needed to be an array and could be a scalar instead.

CENTORI The functions in the CENTORI optimisation library had been written in a generalised form, with no reference to identifiers or data structures in the CENTORI code. All of the data that was operated on was passed into the functions, and so it was not necessary to search the code to see what was accessed and then find out the information about the variables that would be needed to transfer the data to the GPU, as all this was presented in the code of each function. The task of porting the code was further simplified as all of the functions were located in a single source code file. Additionally, none of the kernels called other functions, so there was no need to port functions other than those chosen or make changes to the structure of the code. Although it was not data that was operated on, there were some parameters needed for certain kernels that were declared in one of the Fortran source files. To make these available in the GPU code, a new Fortran function was created to pass the values into a function created in the C code. This then allowed the data to be copied into file scope variables in the C code. The Fortran function was called before any of the kernels were executed, so these parameters were available for use by the GPU code before they were needed. To run the kernels on a GPU, it was necessary to create two functions for each of the original kernel functions. One of these functions was executed by the CPU, and served as an interface with the rest of the Fortran code, taking the input intended for the kernel and using it to launch the kernel on the GPU. The second was the actual kernel code that was executed by the GPU.

3.3 Converting to C

Once all of the code that must be ported to the GPU is known, conversion to a language that can be compiled for GPU execution can begin. As previously described in section 2.3, Nvidia's `nvcc` compiler currently only supports GPU code being written in C-based CUDA. C is one of the most popular languages for HPC applications, however other languages such as Fortran and C++ are also common. It is therefore not unlikely that code may have to be converted into C from another language in order for it to run on a GPU. The effort this requires varies based on how different the language is from C. Fortran 77 and 90, for example, are procedural languages similar to C, and so the conversion is likely to be very straightforward. C++, however, contains features of an object-oriented language such as classes, which have no direct analog in C and so larger changes to the structure of the program may be necessary. It should be considered when the conversion is being performed that it would be desirable to minimise changes to the code, especially if the original CPU-only version continues to be actively developed, so that maintaining the GPU version is not unnecessarily burdensome. When the non-GPU code is written in a language other than C or C++, it will also be necessary to ensure that the two codes written in different languages interface correctly.

Ludwig As Ludwig was already written in C, no such conversion was necessary. It is possible to compile the GPU code with `nvcc` into an object file, and then link this with the object files of the rest of the code compiled with `gcc`, or just by compiling all the files together using `nvcc`. This is not surprising as even when `nvcc` is used on all of the files, it will just send the files that have a .c extension to be compiled by `gcc`. An issue was uncovered when trying to compile the GPU code with `nvcc`, however. The file included several header files from the original code. Although `gcc` didn't find any fault with these files when they were included from the other host code source files, the CUDA compiler complained that the datatype FILE was not recognised in one file. This problem was fixed by including *stdlib.h* in the header file, where FILE is declared. It is probable that `gcc` was including this file automatically, while it must be explicitly included for `nvcc`. Despite being compiled with the flag 'host-compilation=C', which informs the compiler that the host code is written in C (rather than C++), an error was still caused by the identifier 'new' in one of the header files as it was being interpreted as the C++ operation of the same name.

FAMOUS Being a Fortran 77 code meant that FAMOUS was mostly written in uppercase by convention. C, however, is conventionally written in lowercase so the first step in converting the function to C was to make this change. To avoid making an unintended alteration, the conversion was performed on a line-by-line basis so that each line could be checked before being modified to make sure something that was still meant to be uppercase in the C version wasn't accidentally made lowercase. `Vim`, the text editor used, has a feature to convert a line to lowercase. As well as reducing the effort needed and significantly reducing the time needed to complete this task, using an automated feature such as this also decreased the chance of introducing mistakes into the code during the conversion. The next stage was to change the Fortran syntax for accessing an array element, `A(x)(y)`, into the C syntax. It was decided that the best was to accomplish this was by using C preprocessor macro functions. This involved creating a `#define` statement for each of the arrays present in the function, which called another preprocessor macro based on the size of the array. An example of this can be seen in Listing 3.1. For a two dimensional array called trans_d, this would therefore convert `trans_d(x)(y)` into `twoarr(trans_d, x, y)`, which was in turn replaced by code that would access the correct element in C syntax. This code was in fact more complicated than the standard C array syntax `trans_d[x][y]` because the arrays were allocated using the version of `malloc` for GPUs, `cudaMalloc`, and the array identifier is passed to the GPU function without informing it of the number of columns. The elements must thus be accessed using pointer arithmetic. The advantages of this system are that it makes the conversion faster since there is no need to go through the code making changes to the array syntax. Not altering the code is also good because it leaves the code as similar to the CPU version as possible which means that it is easier to keep the two versions synchronised. A final advantage is that it makes debugging less difficult as the code that controls which elements of an array are accessed only needs to be changed in one location rather than throughout the entire function. A complication with converting code from Fortran to C is that Fortran accesses arrays in column-major order whereas C uses row-major ordering. This means that an array is effectively transposed when it is passed between codes in the two languages. A further issue is that Fortran starts array indices from 1 while they start from 0 in C. The preprocessor macros described previously were useful for the last point, as they meant that this issue could be dealt with by simply inserting a minus one on the array indices in one location for each type of array.

Listing 3.1: An example of the preprocessor macro functions used to convert from Fortran array syntax into C array syntax

```
#define trans(A, B) twoarr(trans_d, A, B)

/* The x dimension of the trans_d array is of length
 * npd_profile_d. The loop over the x dimension starts
 * from 0, but y dimension does not, and so must be
 * corrected to start from 0 for conversion to C */
#define twoarr(NAME, A, B) (*(NAME+(B-1)*(*npd_profile_d)+(A)))
```

CENTORI Despite being written in Fortran 90, the optimisation library for CENTORI was already in lowercase so doing this conversion was not necessary. Using C preprocessor macro functions to access array elements had already been used successfully in FAMOUS, as described above, and so was employed again. With this code, however, arrays of the same name were not always of the same size. This is because the kernels were written to be very general and so the input arrays were usually called ‘a’ and ‘b’, for example, and the result array ‘output’, even though sometimes the arrays were three dimensional while in other kernels they were only two dimensional. For this reason it was not possible to have a macro for each array name that redirected to another macro based on what type of array it was. The method used was to just have macros for the various array sizes used and to modify the code so that the appropriate macro was called for the size of the array in each particular case. Accessing array ‘a’ using a(x, y, z) was thus replaced by AT2(a, x, y, z), where AT2 was a macro function that produced correct C syntax to access the desired element of array ‘a’. While this approach no longer provided one of the benefits that was obtained in FAMOUS (avoiding modification to the original code), it still had some effect in reducing the change to the code and continued to provide the advantage for debugging of having the new array access code defined in only one location.

To facilitate the later optimisation stage by reducing the complication of interfacing between different languages, another version of the CENTORI optimisation library was also developed that involved rewriting the functionality of the remaining Fortran code using C. This required creating methods for launching the different versions of each kernel that would be written, measuring the runtime for performance analysis, and verifying that the GPU implementations produced the correct results.

3.4 Decomposing into many threads

The chosen section of code will usually consist of a loop, and possibly nested loops. As there is essentially no overhead associated with creating and destroying threads, and GPUs perform best with many threads, each thread typically executes a single loop iteration. The next stage of the porting process, therefore, is to decide how the code will be decomposed into GPU threads. As it is only possible to have one layer of thread blocks in the z dimension, if the problem domain is 3D then the number of threads per block in the z dimension would probably be set to the number of elements in that direction. Deciding how many threads each block has in the x and y dimensions to give the best performance is difficult. A point to note is that in the currently available products there can be a maximum of 512 threads per block. As well as this limit, there is also a finite amount of memory resources available per SM on the GPU. The more memory each block uses, the fewer blocks can be active concurrently, impairing the GPU’s ability to cover memory latency through time-slicing. Determining the number of threads per block in each dimension can be determined by experimentation to see what configuration results in the lowest runtime.

Ludwig The collision function in Ludwig contains three nested loops to iterate over the elements in the 3D domain. As suggested above, the number of threads per block in the z direction was set equal to the number of elements in this direction. The number of threads per block in the x and y dimensions was initially set to 1. After the ported code had been verified to produce the correct results, a sweep of the possible combinations of threads per block in these dimensions was performed to determine the optimum configuration. The tested values were powers of two from 1 to the maximum number of elements in the x and y dimensions, with the constraint that the total number of threads per block was less than the maximum allowed. These values are likely to give the best results as the number of SPs per SM and the warp size are both powers of two, and if they evenly divide into the number of threads then the wasteful situation of having idle SPs because they don't have a thread to execute, will not occur. To ensure that there are enough blocks of threads to cover the problem domain, the number of blocks in each direction is set equal to the number of elements in that direction divided by the number of threads per block in that direction and the result truncated to an integer, plus one block if the two numbers don't divide evenly. It is necessary to enclose the GPU code within an `if` statement that only allows execution within it if the thread is within the problem domain to prevent any threads from attempting to operate beyond the problem domain boundaries. The results of this can be seen in Fig. 3.1. It is clear that the configuration of threads per block can have a significant effect on performance, with the kernel runtime almost doubling between the fastest and slowest of the configurations tested. The chart also demonstrates that with so many factors involved in determining the runtime for a given thread block configuration, it is difficult to develop a theoretical explanation of the behaviour which would allow the optimal configuration to be calculated without experimentation.

FAMOUS As was previously mentioned, the function from FAMOUS that was ported did not contain one large loop, but instead several smaller loops. The loops consisted of two nested loops to cover the elements in the 2D input arrays. Despite this, there was no requirement that each loop have finished updating every element before the next loop began as in every case updates to an element only relied on elements in other arrays at the same coordinates. This was very appropriate for the GPU many-thread model as performing all of the operations in the function on a single array location can be allocated to a single thread. The number of threads per block in the z direction was set to one as this code only operates in two dimensions. The x and y block dimensions were set in the same way as Ludwig above. The results for various numbers of threads per block can be seen in Fig. 3.2. The variation was again quite large, however this time the results behaved in a more predictable manner. The execution is clearly faster when the threads are arranged in the y dimension, rather than the x dimension. The fastest block configuration, (1, 16), gives a GPU kernel execution time of 0.025 ms, which is a speed-up of 20 times over the CPU execution time of 0.51 ms. When memory transfer between the GPU and CPU is also considered, however, the total GPU runtime increases to 0.52 ms. Memory transfer therefore erases the performance improvement obtained through faster kernel execution. It is likely that if more of this code were ported to a

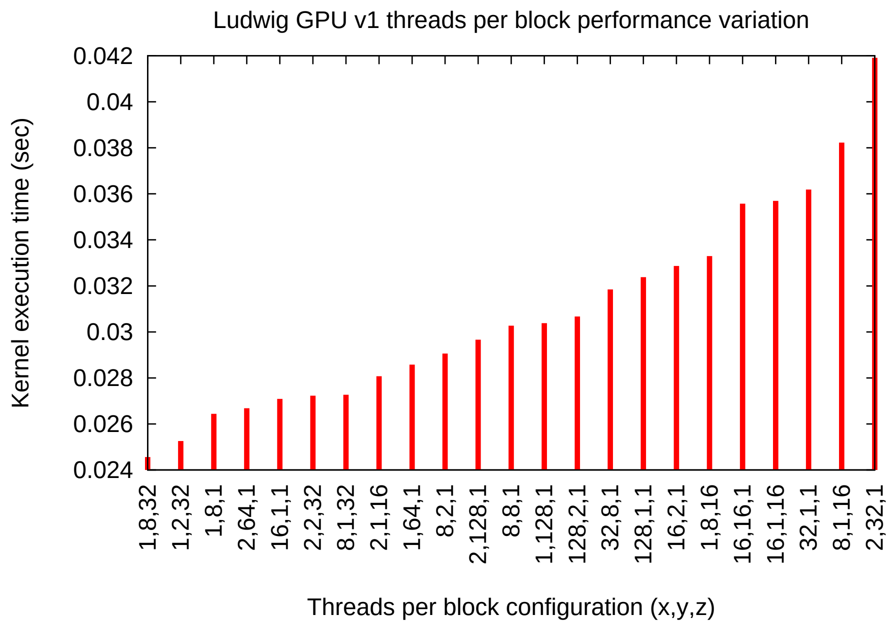


Figure 3.1: The kernel runtime of Ludwig GPU v1 for a selection of the tested threads per block configurations

GPU, then the amount of calculations performed per data element transferred could be increased substantially, resulting in a better total speed-up.

CENTORI Each of the kernels in the CENTORI optimisation library contained three nested loops to iterate over the elements in the arrays. As each of the loop iterations was independent, as with the other codes, the decomposition of the loops into GPU threads was performed in the same way as previously described.

3.5 Debugging

Most of the time during the porting stage of the project was spent on debugging. This was because doing this is much more difficult and time consuming than on a CPU. One of the main ways to debug code on a CPU is to examine the value of variables at certain points in the code to see where it is going wrong. This is normally done by putting a print statement at the locations where the value wants to be checked. Unfortunately this method cannot be used on a GPU because it is not possible to call host functions, such as a print statement, from the GPU code. This means that the only way that an approach such as this can be used to it create another data structure similar to the one that it is wished to examine, and copy the real data into the new variable at the point where the value is to be checked. To achieve this two new pointers must be declared: one that will be the new GPU variable, and another that will store the data that is copied back onto

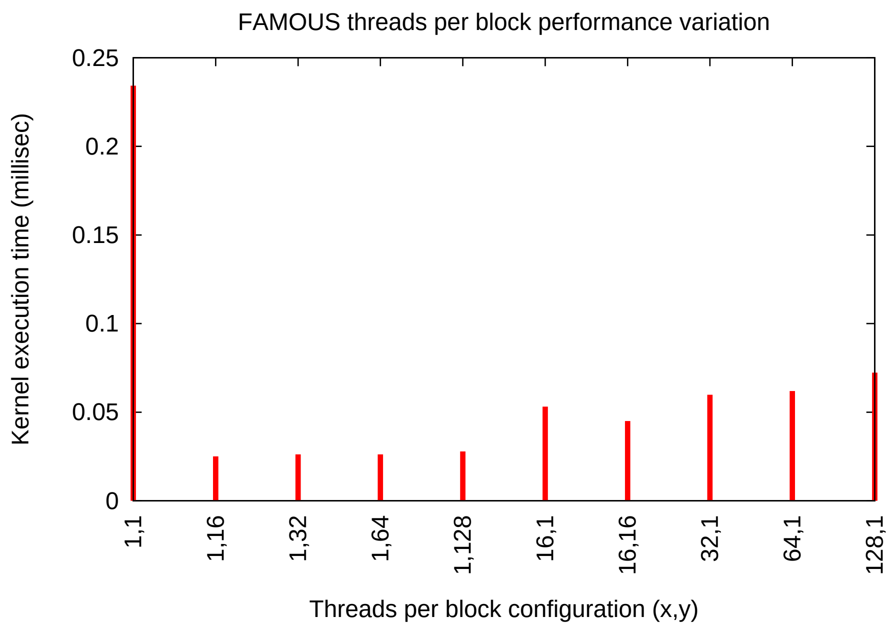


Figure 3.2: The kernel runtime of FAMOUS with different threads per block configurations

the CPU from the GPU debugging variable. Both of these variables must be allocated enough memory to store the amount of data that will be examined. The GPU pointer must be passed into the GPU kernel so that it can be accessed from the GPU code. A statement to copy the data at the desired point in the code must be inserted. The data must then be copied from the GPU debugging variable into the CPU version of it that was created. Finally the data now stored in the CPU variable must be printed. As well as being tedious and very time consuming, this approach also has a number of other difficulties. The first is that checking the value of a variable at several locations in a code, which is what is often desired so that changes to it could be tracked, is even more difficult as either several additional GPU debugging variables must be created along with their CPU counterparts, or the original could be made several times the size of the original data structure and complicated book-keeping used to store the value at different points in the code into different locations in the memory allocated to this debugging variable. Furthermore, when this type of debugging is being performed in a serial CPU code, values can be printed in the order in which they occur. For example, the values of several different variables in each loop iteration can be printed. This is very useful for understanding why a code is not producing the correct results. Producing this type of output when the debugging is being performed on a GPU in the way described above is difficult. This was the primary method of debugging used during the dissertation.

It is possible to run GPU code in device emulation mode which attempts to emulate a GPU on the CPU. There are several ways in which the device emulation version of the code will behave differently from the GPU version. One way in which it is different is that it allows host functions to be executed from the GPU code. This means that it is possible to execute print statements. Another difference is that it is possible to use `gdb`, the GNU Debugger. This is a tool which allows the execution of code to be halted at any location of the source code desired and the values of variables that are defined at that point of the execution to be printed. The device emulation mode differs in other ways which mean that it is often not useful for debugging, such as allowing GPU kernels to access host data. It is thus frequently the case that code will execute differently on the GPU and in device emulation mode and so errors that occur when the code is being run on the GPU may not occur when it is running on the CPU.

A new method for GPU debugging was introduced in CUDA Toolkit v2.2. The CUDA-GDB Debugger allows `gdb` to debug code executing on a GPU in a similar way to CPU code. This means that the value of variables can be checked at any point in the code even when it is running on the GPU. This obviates the need to do the time-consuming method of copying data into data structures defined for debugging, as was described above. It was not possible to use this method as version 2.2 of the Toolkit was only released shortly before the project began and so had not yet been installed on the GPU systems used.

The difficulties encountered during the debugging process were compounded by the poor error and warning capabilities of the CUDA compiler, `nvcc`. Situations where the `nvcc` compiler made unwanted changes to the code without warning, gave poor error messages, and produced no error messages when it would be expected that it should, were encountered, some of which were documented above. There is currently no equivalent to the `-Wall` flag, or any other flag which would increase the number of warning messages produced during compilation. The compiler is still relatively new and so such problems are expected. Without the guidance of good error and warning messages, locating the source of problems in the code requires significantly more debugging, which, as was presented above, is a very time consuming process on a GPU.

Chapter 4

Optimisation

In order to efficiently use GPU resources, it is usually necessary to perform significant optimisation. The ideas first presented in the earlier example (section 2.3.1) will be used for Ludwig and the CENTORI optimisation library. These include ensuring that Global Memory accesses are coalesced, reducing unnecessary accesses to Global Memory, and reducing register usage.

The results presented in this chapter are based on measurements made on Intel Nehalem CPUs and Nvidia Tesla C1060 GPUs at Daresbury Laboratory.

4.1 Ludwig

The initial, unoptimised GPU port of Ludwig already delivers a small performance improvement in the execution of the kernel. Fig. 4.1 shows that the kernel executes about 2.75 times faster on one GPU than on one CPU for a problem size of $128 \times 128 \times 128$. It is obvious, however, that the type 2 scaling (increasing the number of GPUs, but keeping the problem size fixed) is worse than the original CPU version of the code. When 32 GPUs are used, the GPU code executes at about the same speed as the CPU version with 32 CPU cores. It is believed that this is caused by two issues. The first is that the GPUs are being under utilised. Each of the GPUs used during this experiment can have up to 184320 threads simultaneously active, however when 8 GPUs are used, the problem size received by each GPU can only be decomposed into 65536 threads – 35% of the available capacity. Increasing the number of GPUs therefore results in little decrease in the time needed to complete the kernel, unlike the case when additional CPUs are used. The other issue is that this code requires the use of a very large number of registers per thread: 52. This limits the number of threads that can be simultaneously active, and also requires that some of the register values be stored in Local Memory which is located in Device Memory and so is very slow. It is possible that another cause of the better CPU scaling is that more of the problem fits into the CPU cache when the problem size is reduced as the number of CPUs used increases. As GPUs do not have

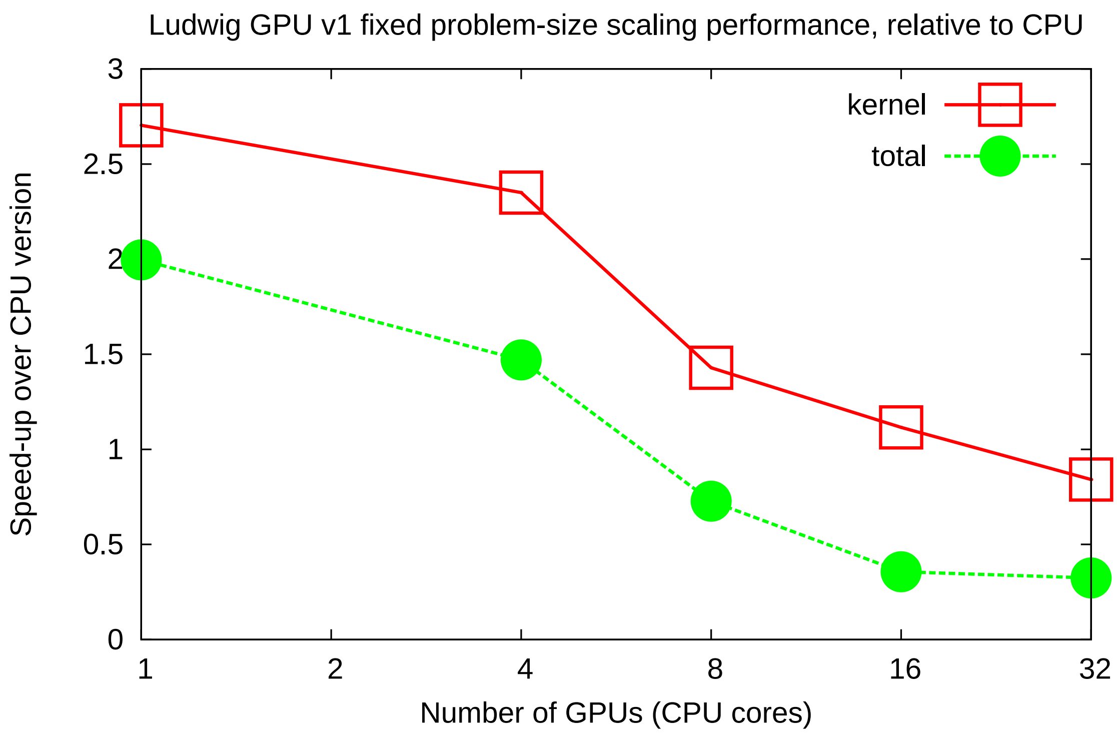


Figure 4.1: Speed-up obtained by version 1 of the GPU implementation of a section of Ludwig for varying number of GPUs (compared to an equivalent number of CPU cores) for a problem size of $128 \times 128 \times 128$. Data is shown for the case when only the execution of the kernel is timed, and the case when other GPU-related operations, such as copying data to and from the GPU, are included.

a cache similar to that present in CPUs, and this version of the GPU code does not use any of the on-chip forms of memory, this effect is not present in the GPU case.

The ‘total’ time presented on the graph is that of kernel runtime plus the time taken by copying input data onto the GPU and copying the changed data back to the CPU. It is currently necessary to transfer this data in order to use the GPU, and so this is a more accurate measure of the performance improvement that could be obtained by using the GPU. In section 4.1.7 an argument will be presented to demonstrate that it should be possible to greatly reduce the amount of data transferred between the CPU and GPU during each iteration, and so a performance improvement close to that of the kernel execution could be achieved.

The type 1 scaling (increasing the problem size in the z-dimension while keeping the number of GPUs constant) performance of the GPU is compared to that of the CPU version in Fig. 4.2. The kernel execution time speed-up appears to be independent of the size of z. The GPU used allowed a maximum of 64 threads in the z-dimension, although it was actually limited to 32 in order to allow a higher number of threads in the x and y dimensions (the total number of threads per block must be less that 512).

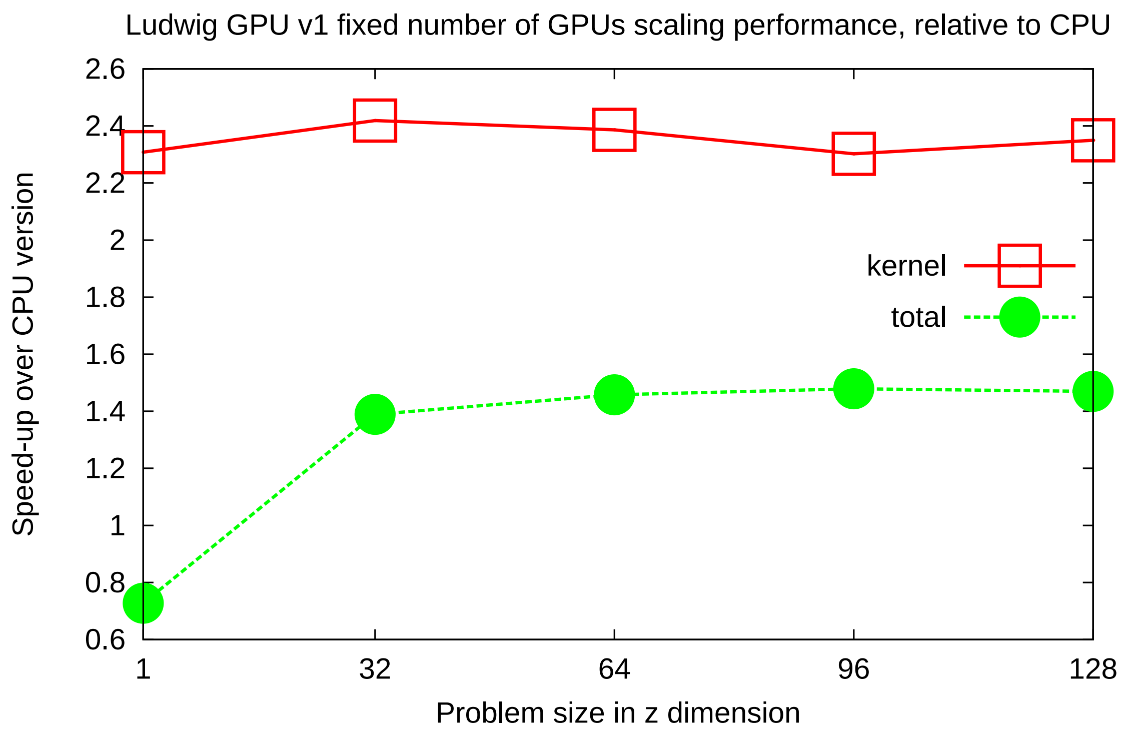


Figure 4.2: Speed-up obtained by version 1 of the GPU implementation of a section of Ludwig for varying the problem size in the z-dimension (with problem size in the x and y dimensions fixed at 128) on 4 GPUs (compared to 4 CPU cores). Data is shown for the case when only the execution of the kernel is timed, and the case when other GPU-related operations, such as copying data to and from the GPU, are included.

Therefore, problem sizes with z higher than 32 cannot use any more threads, and so the utilisation of the GPU is the same. This results in the performance compared to the CPU being the same for these problem sizes. It is surprising that the speed-up is the same for the $z = 1$ problem size as the GPU utilisation would be 32 times less in this case. It is likely that the decrease in computing power is being compensated by improved memory access speeds due to a reduction in the demand for Global Memory bandwidth.

An additional chart, showing the scaling behaviour as the problem size in the x dimension is varied, while the sizes of the other dimensions are kept fixed, is available in Appendix B.1.

The speed-up of the total GPU runtime increases with the problem size. This is probably because constant time overheads are responsible for a smaller percentage of the runtime.

To measure the changes produced by the various optimisations implemented, the times required for each of the GPU-related operations were recorded for each version, and are presented in Fig. 4.3. For the configuration of problem size and number of GPUs that was used for this graph, the memory allocation time would be the most significant

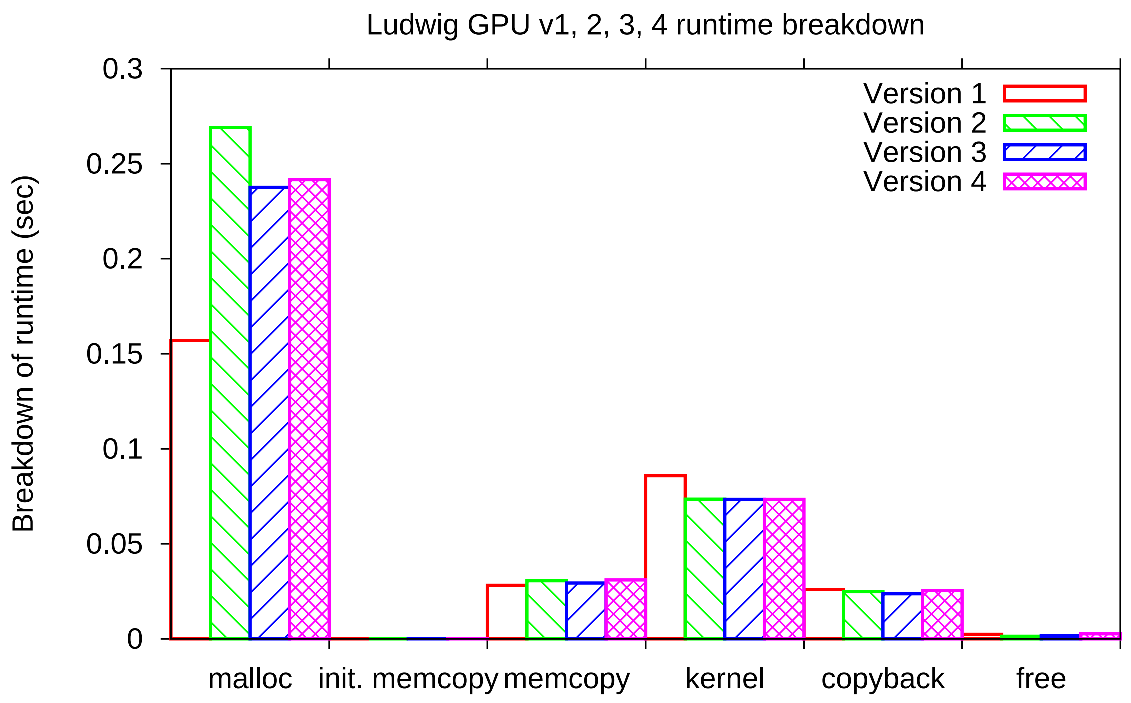


Figure 4.3: Breakdown of the runtime spent on GPU-related operations for one iteration of versions 1, 2, 3, and 4 of the GPU implementation for problem size $128 \times 128 \times 32$ on 4 GPUs. Note that the 'malloc', 'init. memcopy', and 'free' operations are only performed once per application launch, while the other operations are performed every iteration.

element of the GPU runtime if only one iteration were executed. As it is assumed that for production runs many iterations would be executed, the memory allocation, which is only performed once per application launch, would be irrelevant. Of the operations that are performed every iteration, the kernel execution was responsible for approximately half of the runtime.

4.1.1 Storing parameters in Constant Memory

Version 2 of the GPU code represents the first optimisation attempted. It was clear that a large amount of data was needed by the kernel, as shown by the time required for the memory transfers between the CPU and GPU. This implied that the kernel was likely to be memory-bound, which means its performance is limited by the speed at which data can be loaded from and stored to memory. Reducing unnecessary Global Memory loads was therefore an obvious target for improving performance. This was achieved by loading various parameters needed by the kernel into Constant Memory rather than Global Memory. As previously described, Constant Memory is a cached memory space with a broadcast mechanism, making it ideal for storing such data. It is clear from Fig. 4.3 that a noticeable improvement in kernel performance is obtained from this optimisation. It is also apparent, however, that the memory allocation time has significantly increased. As previously argued, this is not very important as it is only performed once.

4.1.2 Reducing unnecessary CPU-GPU data transfer

The next optimisation aimed at reducing memory transfer time by only transferring data that changed between kernel invocations to the GPU each iteration. Other data was only transferred once, during a new ‘initial memcopy’ phase. In the current version of the code, memory must be transferred to the GPU every iteration, so reducing the time required for this is just as beneficial as reducing the runtime of the actual kernel. Unfortunately it is evident from Fig. 4.3 that this optimisation had little effect. The amount of data that was determined to not change between iterations was so small that the time taken to transfer it is not visible on the graph.

4.1.3 Reducing number of kernel launches

Reducing kernel execution time was again the aim of the optimisation implemented in version 4 of the GPU code. As was previously mentioned, the maximum number of threads in the z dimension was limited to 32. This meant that for problem sizes that were greater than 32 in the z dimension, it was necessary for each thread to operate on multiple elements in the z dimension. In previous versions of the code, this was achieved by launching the kernel multiple times, with the threads operating on elements with higher z values each time. The problem with this approach is that there is a cost associated with launching a kernel. Version 4 of the code therefore moved this loop over the z dimension inside the kernel, so that it could be achieved with only one kernel launch per iteration. The advantage of this optimisation is not visible in Fig. 4.3 as the problem size used only has 32 elements in the z dimension, so this multiple elements per thread procedure is not needed. This optimisation results in the use of 10 extra registers per thread, however this does not appear to decrease performance.

To guide further optimisation, the kernel was divided into 15 roughly equally sized segments, and each one was timed. This gave a greater understanding of what operations in the kernel were consuming the most runtime, allowing optimisation efforts to be directed towards issues that had the greatest potential for delivering a good performance improvement. The results, which can be seen in Fig. 4.4, show that four of the segments (1, 11, 13, and 15) are responsible for significantly more runtime than average. This was expected as these four segments each cover a section of code where the `site` array is accessed. This is one of the two outputs of the section of code that was ported, and is by far the largest data structure present as it contains 38 doubles for every element. Reducing accesses to this array, which is stored in Global Memory, is therefore likely to produce a notable improvement in performance.

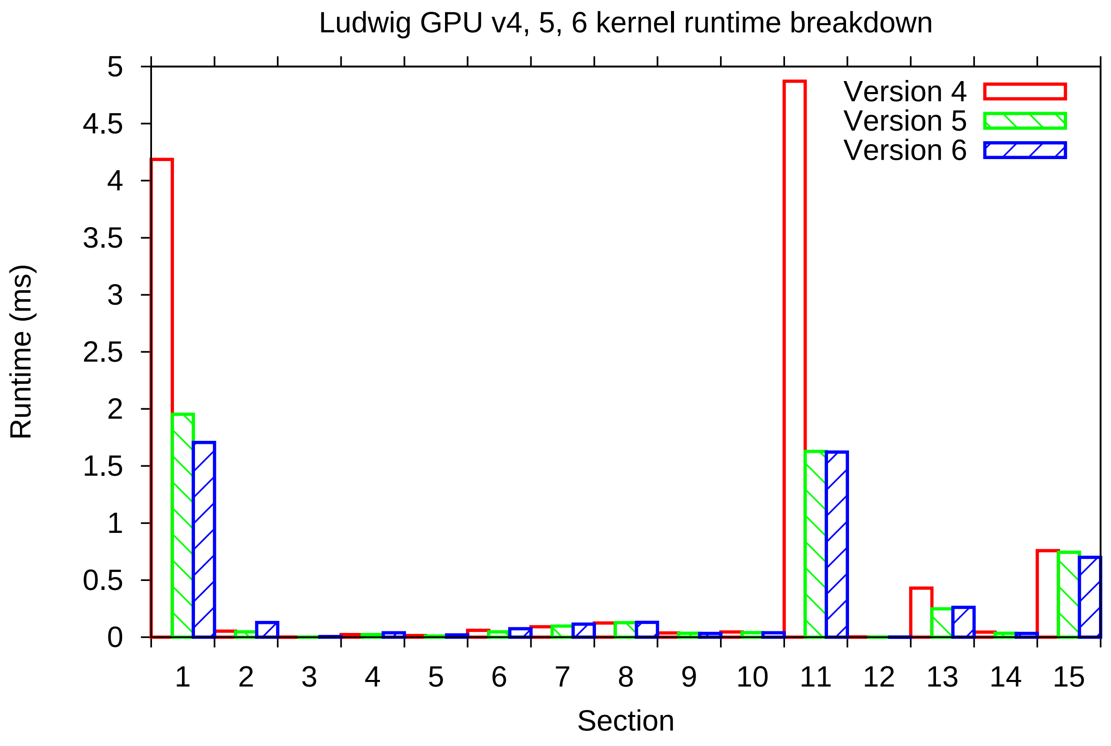


Figure 4.4: Time per iteration spent in different sections of the kernel by a single thread for versions 4, 5, and 6

4.1.4 Eliminating unnecessary Global Memory accesses

Version 5 of the GPU code eliminates several accesses to the `site` array in Global Memory by storing intermediate results in registers. This is achieved by declaring new arrays local to each thread, to store the data of the `site` array for that thread. The data is copied from Global Memory once at the start of the kernel, and is written back to Global Memory at the last point in the kernel where the data is accessed. The reduction in the time required for a single thread to execute the kernel by performing this is very clear from Fig. 4.4. It is important to note, however, that this was achieved with a significant cost in terms of register usage. Storing an additional 38 doubles per thread in registers will greatly reduce the number of threads that can be simultaneously active. Because of the large number of registers being used, each thread uses 1.7KB of Local Memory. This is detrimental for performance, but is necessary in order to achieve the desired optimisation.

4.1.5 Restructuring data to improve coalescing

While a large improvement in performance had already been obtained by altering access to the `site` array, it was believed that it may be possible to increase it further. In all of the versions presented so far, access to the `site` array was not coalesced. This is because the problem size was a multiple of 16 elements in each dimension, however a single element halo was added to this on the boundaries. The result of this is that the starting address of the segment of the array processed by each thread block would not be a multiple of sixteen elements from the beginning of the array, which means that multiple memory accesses would be needed to obtain the data required. Version 6 of the GPU code attempted to correct this by restructuring the `site` array so that accesses could be efficiently coalesced. This involved removing the halo regions in the restructured copy, as they were not used in the kernel, and using the `cudaMallocPitch` and `cudaMemcpy2D` operations described in section 2.3 to ensure that each XY plane would be correctly aligned for coalescing even if the problem domain wasn't a multiple of sixteen elements. Another alteration necessary to improve coalescing was to rearrange the data so that neighbouring threads accessed neighbouring array elements. In the original structure of the array, the 38 doubles related to each element were contiguous in memory. This meant that neighbouring threads would always be accessing data in memory 38 doubles apart. Changing this so that the first double precision values for all of the elements in the problem domain formed a contiguous block in memory, followed by a contiguous block of all of the second double precision values, and so on, meant that when all of the threads in a block read from the `site` array, they would be reading from a contiguous chunk of memory. A further change made was to rearrange the data in column-major order (i.e. neighbouring elements in the x dimension were contiguous in memory) rather than the original row-major order (neighbouring elements in the z dimension were contiguous). This was to ensure the arrangement of the data corresponded more closely to the arrangement of threads in a thread block, which are in column-major order. A visualisation of this restructuring is presented in Fig. 4.5.

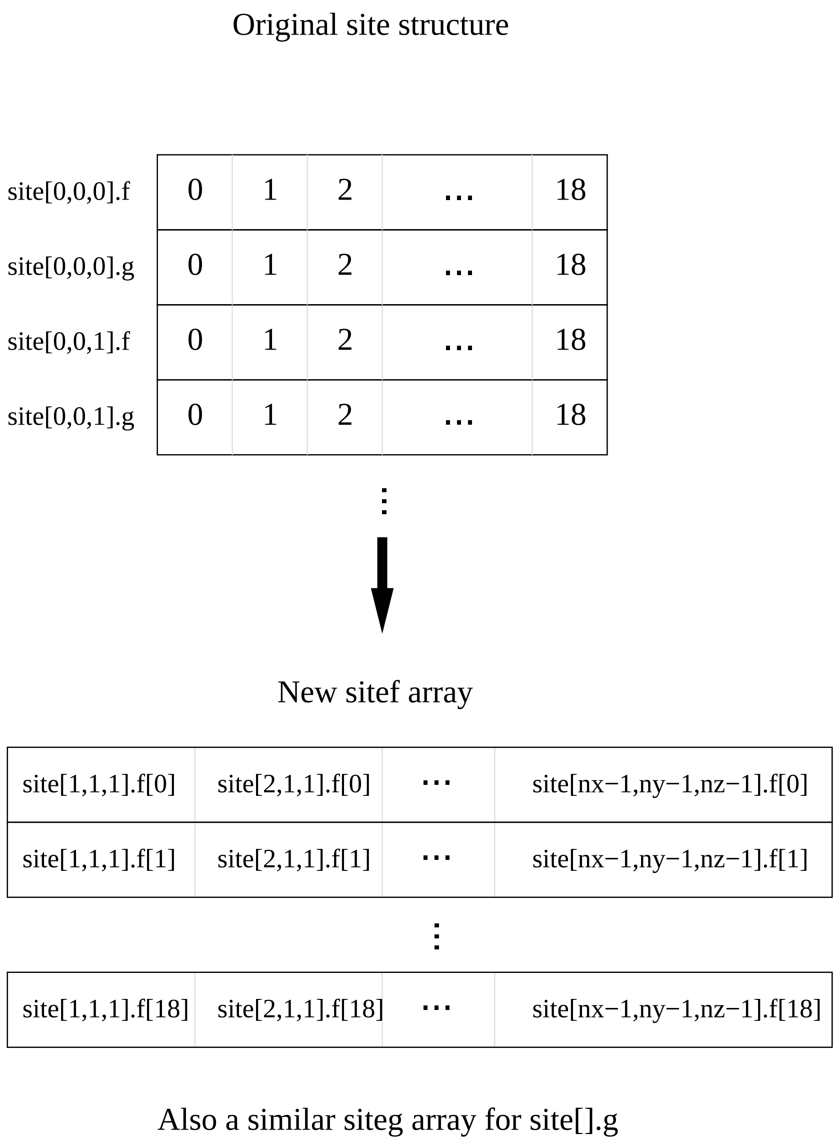


Figure 4.5: The modification to the site array structure to improve memory access coalescing. The array is split into two: sitef and siteg (not shown). The situation shown is for the D3Q19 case, when 19 velocities are stored for every domain element of each fluid.

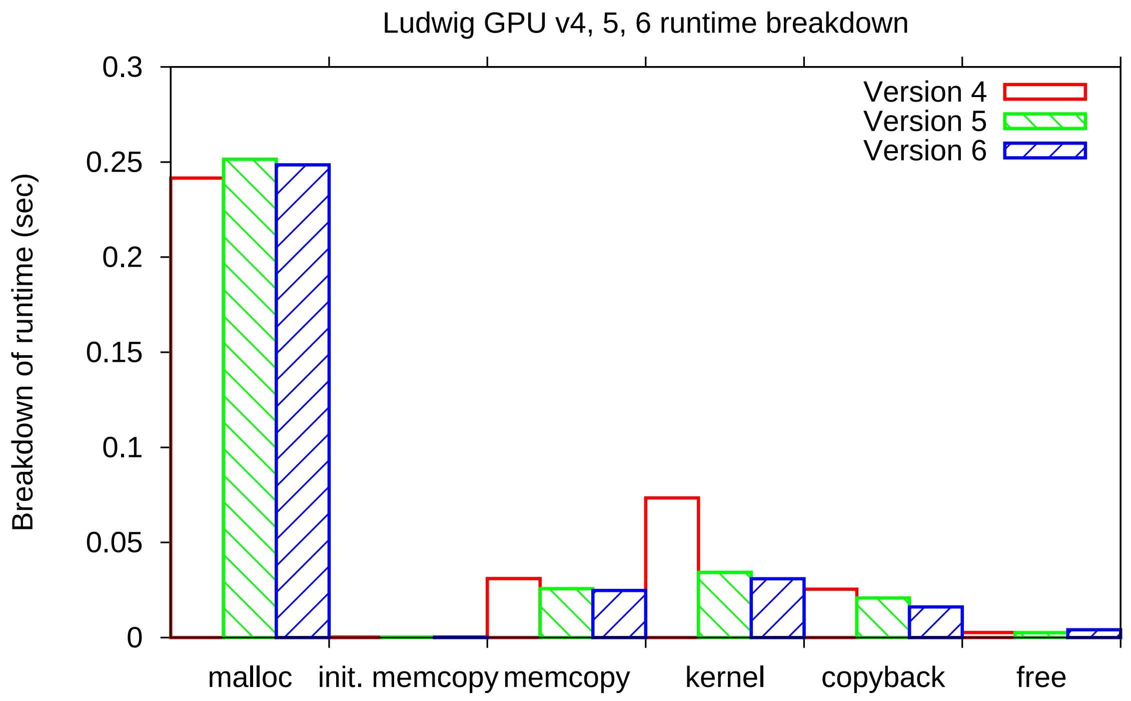


Figure 4.6: Breakdown of the runtime spent on GPU-related operations for one iteration of versions 4, 5, and 6 of the GPU implementation for a problem size of $128 \times 128 \times 32$ on 4 GPUs. Note that the ‘malloc’, ‘init. memcopy’, and ‘free’ operations are only performed once per application launch, while the other operations are performed every iteration.

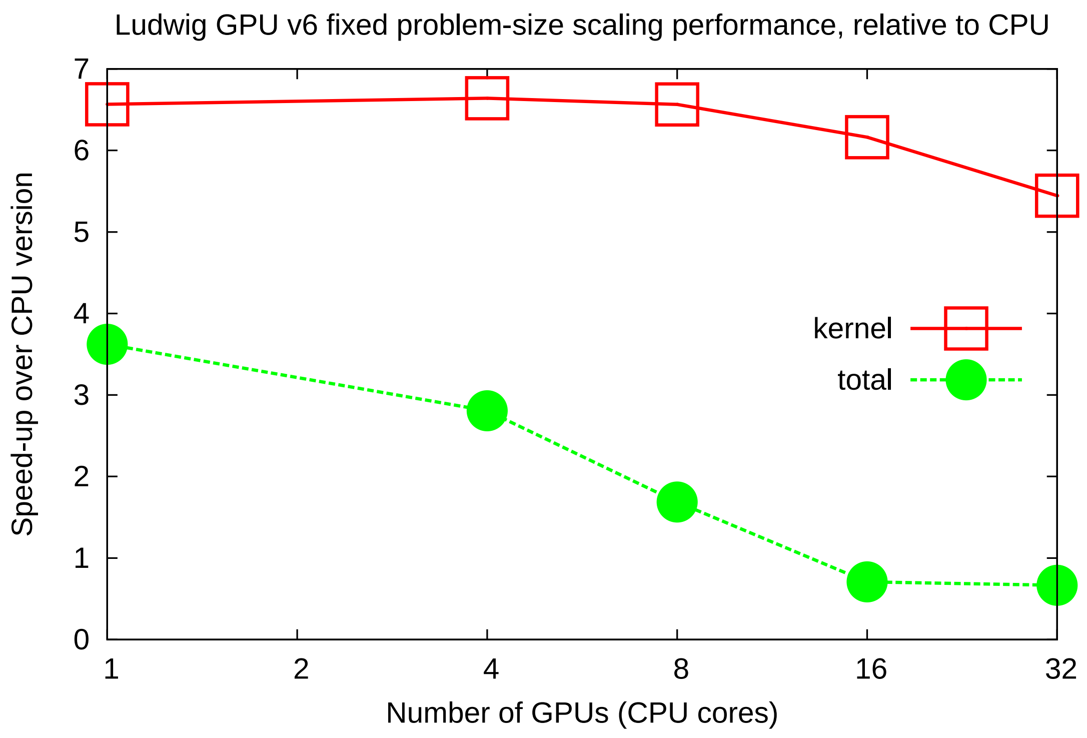


Figure 4.7: Speed-up obtained by version 6 of the GPU implementation of a section of Ludwig for varying number of GPUs (compared to an equivalent number of CPU cores) for a problem size of $128 \times 128 \times 128$. Data is shown for the case when only the exection of the kernel is timed, and the case when other GPU-related operations, such as copying data to and from the GPU, are included.

4.1.6 Final results

The scaling performance of version 6 of the GPU implementation of Ludwig is presented in Fig. 4.7 and Fig. 4.8.

GPUs were designed to use single precision data. This is most apparent in the higher peak performance (over ten times that of double precision). Converting a code to only use double precision where it is necessary therefore has the potential to be extremely beneficial for performance. Performing this would require a very thorough understanding of the functioning of the code. While the results produced are unlikely to be useful, the entire code was converted to single precision to investigate how much of a speed-up could be obtained. The speed-up of the GPU code when compared to the CPU (both using single precision) can be seen in Fig. 4.8. The total speed-up is about twice as fast as the double precision version, which was expected since this measurement is dominated by the memory transfers between CPU and GPU, which will be halved with single precision. It was expected that the kernel speed-up would be at least twice as fast. With a memory-bound kernel, halving the size of the data that needs to be transferred should, one would presume, halve the runtime. The greatly improved computational

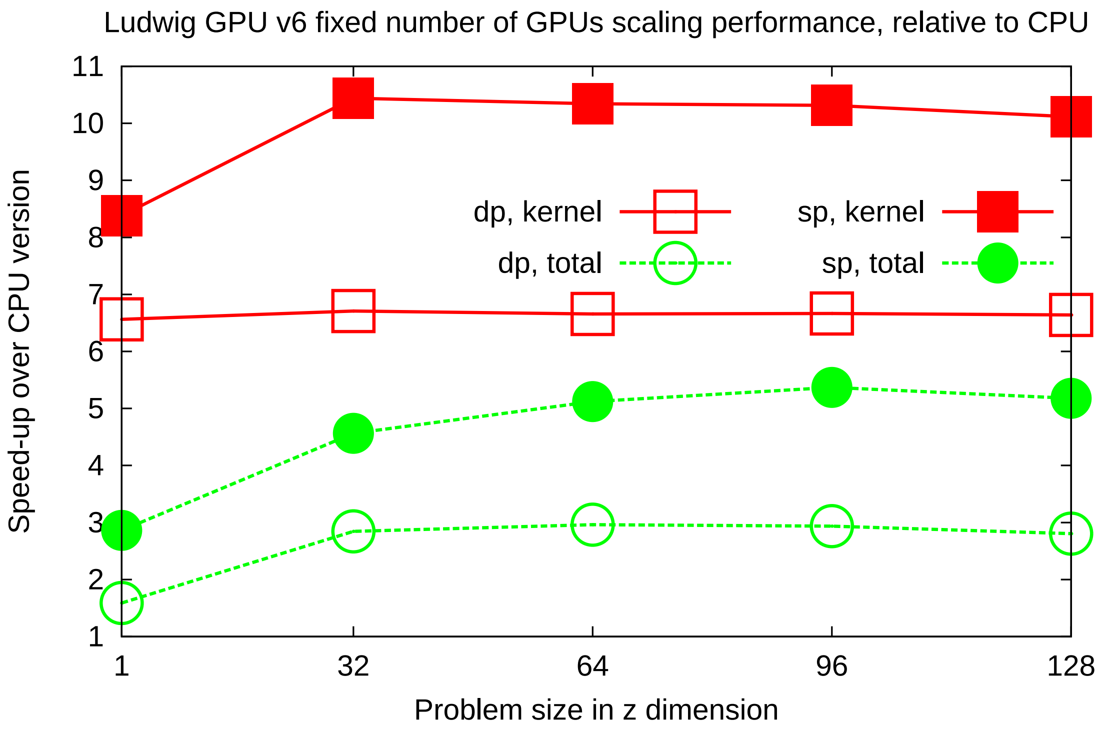


Figure 4.8: Speed-up obtained by version 6 of the GPU implementation of a section of Ludwig for varying the problem size in the z-dimension (with problem size in the x and y dimensions fixed at 128) on 4 GPUs (compared to 4 CPU cores). Data is shown for the case when only the execution of the kernel is timed, and the case when other GPU-related operations, such as copying data to and from the GPU, are included.

power of the GPU for single precision might also have increased the performance a bit further. The approximately 1.6 times increase in speed-up is therefore quite difficult to explain. It is possible that a substantial part of the kernel runtime is caused by the Global Memory latency, which would not be changed by switching to single precision.

Even without further work, the 3 times total speed-up obtained for Ludwig is quite a good result. Even so, the following section suggests some further improvements that may permit a higher speed-up to be obtained in the future.

4.1.7 Further work

It is believed that several further optimisations are possible, and would be likely to improve the performance obtained by implementing Ludwig on a GPU.

One of these optimisations would be to modify the code so that double precision was only used when it was needed. For certain calculations, for example, it might be possible to obtain useful results by only using double precision for certain sections of the algorithm, and single precision for the rest. An example of this is discussed in [3]. As previously stated, this would probably be best performed by someone who is very knowledgeable about the code functioning, such as the code author.

Another possible optimisation could be obtained by rewriting the current GPU implementation so that the problem domain is treated as a one-dimensional vector when it is decomposed into threads, rather than a three-dimensional matrix. This would avoid the problem of being restricted by the number of threads in the z dimension. Another small advantage of this approach would be to make choosing the optimal threads per block configuration less expensive. Currently finding the number of threads per block in the x, y, and z dimensions requires many runs of the code with all of the possible configurations for each problem size. With only one variable to tune, this could be determined much more quickly.

The most important optimisation that could be performed to improve performance of the GPU implementation would be to port more of the code to the GPU. The main advantage that this would bring would be to greatly reduce the amount of data that needs to be transferred between the CPU and GPU during each iteration. In theory, only the halo data needs to be copied back to the GPU so that it can be passed to GPUs working on neighbouring sections. For a $128 \times 128 \times 32$ problem domain divided among 4 GPUs, for example, the size of the halo data for the `site` array stored on each GPU would be 5 MB, which is only 11.5% of the full `site` array stored on each GPU. Eliminating the non-halo data from the transfer to and from the GPU would reduce the amount transferred by 76 MB. Based on the results of Nvidia's GPU bandwidth test, which indicates that the memory transfer speed between the CPU and GPU used for this investigation was approximately 3GB/s, this would reduce the total time per iteration by about 22.8 ms. As the measured runtime in the final GPU version produced for this problem size and number of GPUs was 76ms (using the memory transfer times for version 5, since the halos of the `site` array were not transferred in version 6), this would result in the total speed-up increasing by 70% from 2.6 times to 4.5 times.

4.2 CENTORI Optimisation Library

As with Ludwig, moderate speed-ups were obtained in the kernel execution time for most kernels, as seen in Fig. 4.9. Also similar to Ludwig is the fact that the total GPU runtime is not very impressive – for every kernel the performance is about half that of the original CPU version. Note, however, that for this code it was decided to include the memory allocation and freeing in the 'total' time as, unlike with Ludwig, it is unlikely that each kernel will be executed many times consecutively, and so changes in how the memory is allocated may be necessary. It is probable that in the CENTORI code several kernels could be executed together without the data being modified on the CPU in between, and so memory allocation, transfer, and freeing would only need to be performed once, meaning that a speed-up closer to that of the 'kernel' results could be obtained. Even if this is not the case, it may not be necessary to allocate and free memory for each kernel, as data structures of the same size might be used in different kernels and so allocated space could be reused. Alternatively, with 4 GB of memory per GPU on the model used for this investigation, there may be enough memory to allow for another solution which would be to allocate memory for the different kernels once, and then there wouldn't be a need to allocate and free every time the kernel is called during the application.

To understand the relationship between the 'total' and 'kernel' speed-up results more clearly, the percentage of the GPU runtime spent on executing each of the kernels is shown in Fig. 4.10. These results are almost a perfect complement to the kernel speed-up; the kernels with low kernel-only speed-up spend a high percentage of their runtime executing the kernel. This can be interpreted as indicating that the reason why some kernels are slower is because they have lower performance per data element. This conclusion is arrived at because the time required for the non-kernel GPU operations is directly proportional to the data size (as demonstrated in Appendix A). When the kernel consumes a higher percentage of the total runtime it therefore means that it is taking a longer time to process each data element than other kernels. Another way of visualising how the total GPU runtime is spent is presented in Fig. 4.11.

4.2.1 'a': Restructuring data to improve coalescing

Improving Global Memory access coalescing was an obvious choice for the first optimisation, since every kernel contained non-coalesced accesses to at least one array of a type that will be referred to as 'AT1'. These are arrays of dimensions $nx \times ny \times nz \times 3$, with a three component vector for each domain element. The data was structured so that the three double precision values associated with each element were contiguous

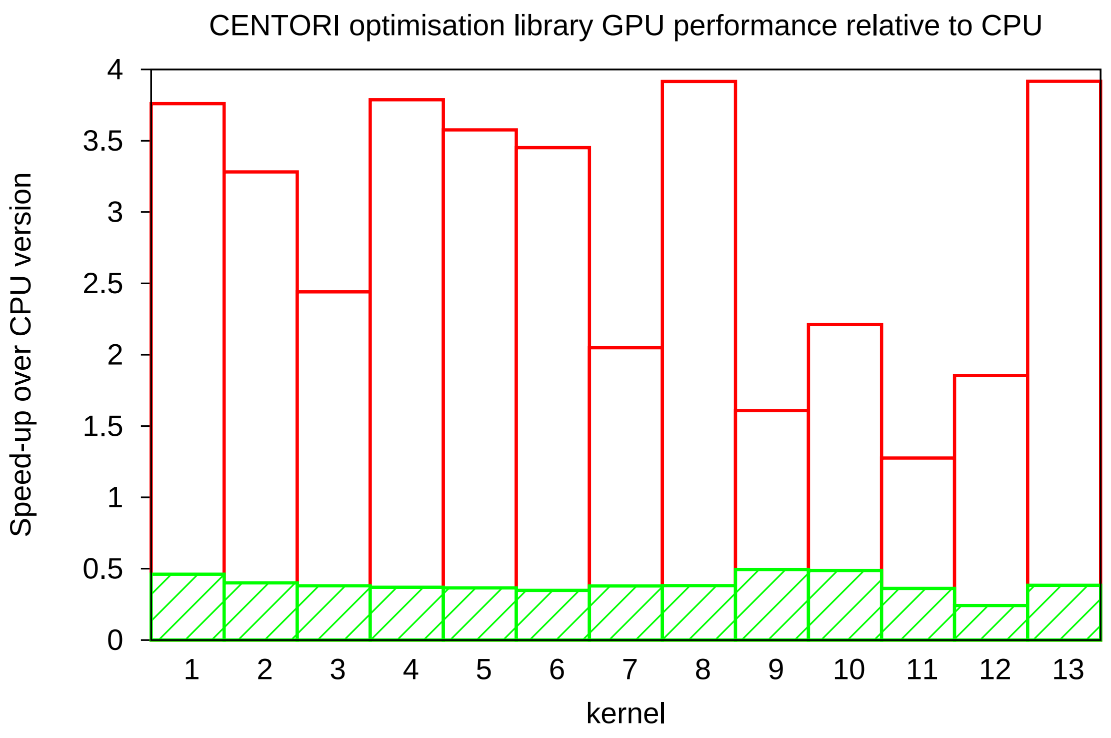


Figure 4.9: Speed-up obtained by porting CENTORI optimisation library kernels to GPU without further optimisation. The results show the case where just the time to execute the kernel on the GPU is considered, and the case where the time for other operations such as copying data to a from the GPU is included.

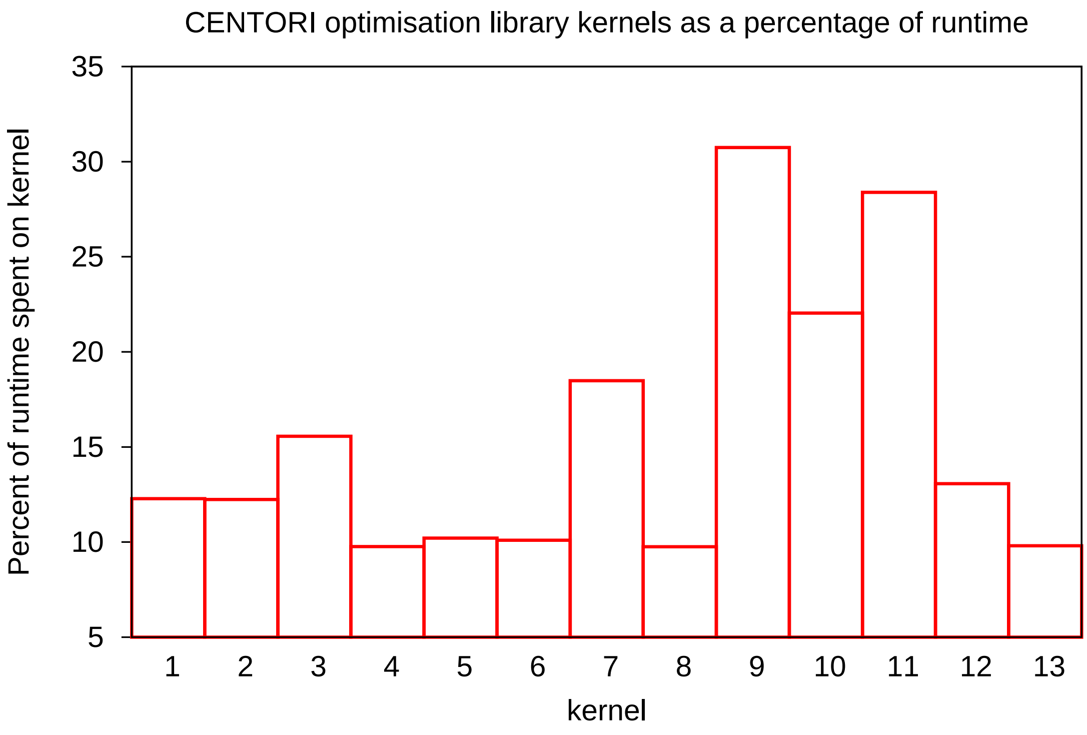


Figure 4.10: The percentage of the time spent on GPU operations that is consumed by the kernel

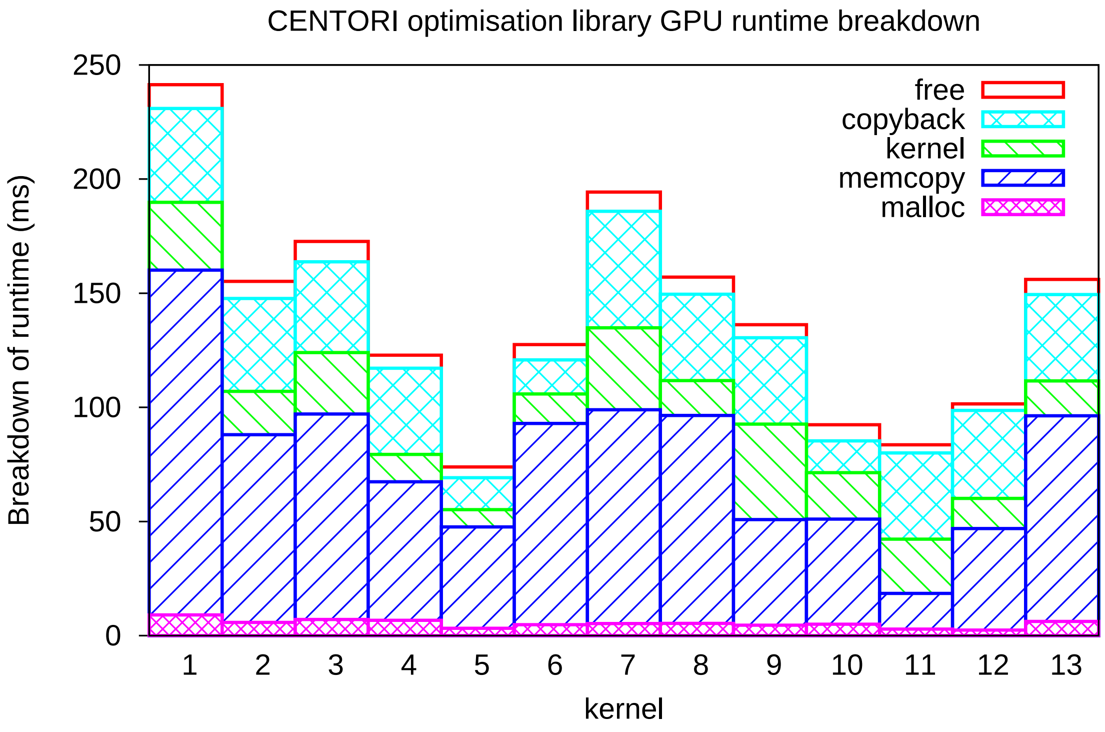


Figure 4.11: A breakdown of how the time spent on GPU operations is divided among allocating memory on the GPU, copying the input data to the GPU, executing the kernel, copying the results back to the CPU, and freeing the memory on the GPU.

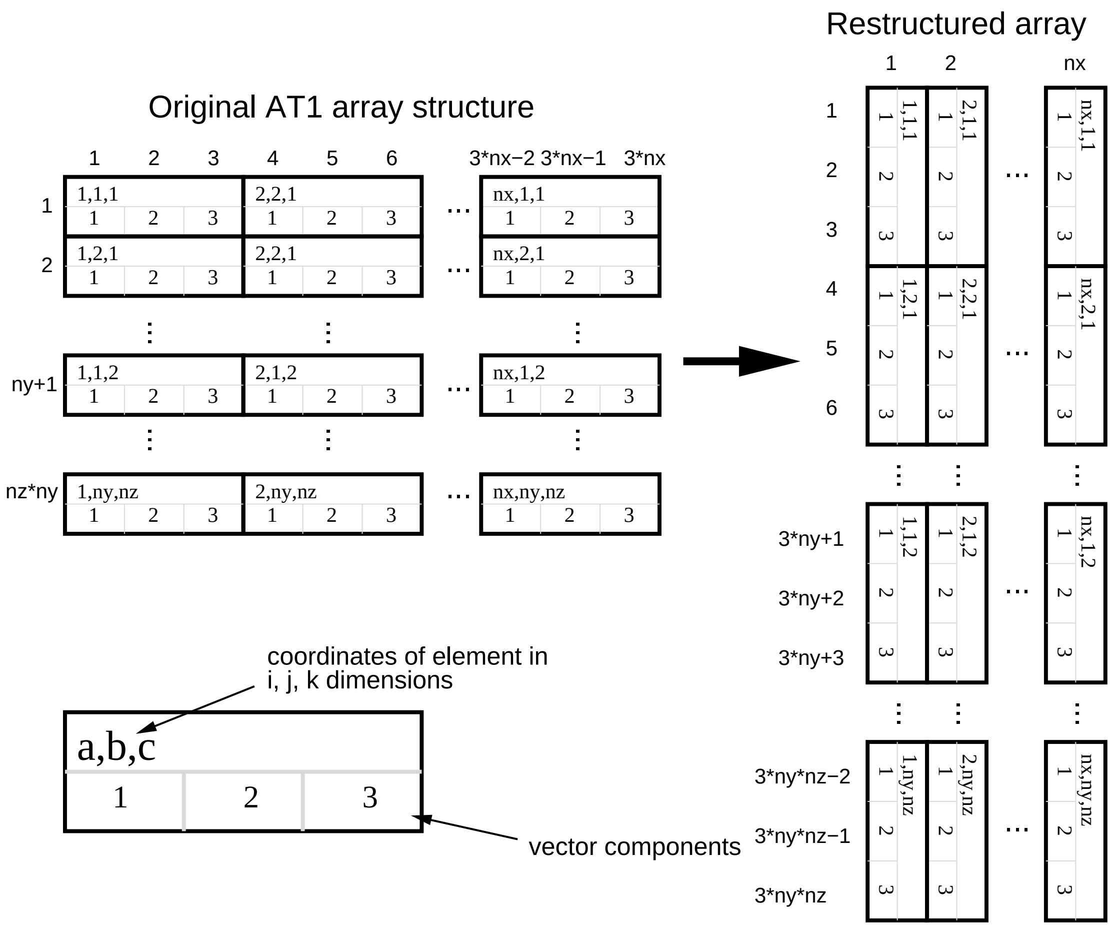


Figure 4.12: A diagram of how AT1 arrays are restructured in the version ‘a’ optimisation. The data is stored in memory using row-major order.

in memory. This structure is not ideal for GPU memory access patterns, since neighbouring threads in a thread block will all access the same component of their individual vectors simultaneously, which means that they will always be reading data three double precision values apart. The result of this is that two thirds of the data loaded from Global Memory for each AT1-type array access will be discarded, as a block of memory containing $3 \times$ (number of threads per block) doubles will need to be loaded each time the threads read one of their vector components.

The solution was to restructure the AT1 arrays so that each of the vector components was next to the equivalent components of the neighbouring threads in memory. A visualisation of this restructuring is visible in Fig. 4.12. The dramatic effect this had on performance is clearly shown in Fig. 4.13. As expected, most of the kernels show a speed-up almost three times higher than that of the unoptimised GPU version.

Input data must be restructured before the kernel launch. If the output array is an AT1 array then a reverse restructuring must be performed on it after the kernel so that it is in the original AT1 form.

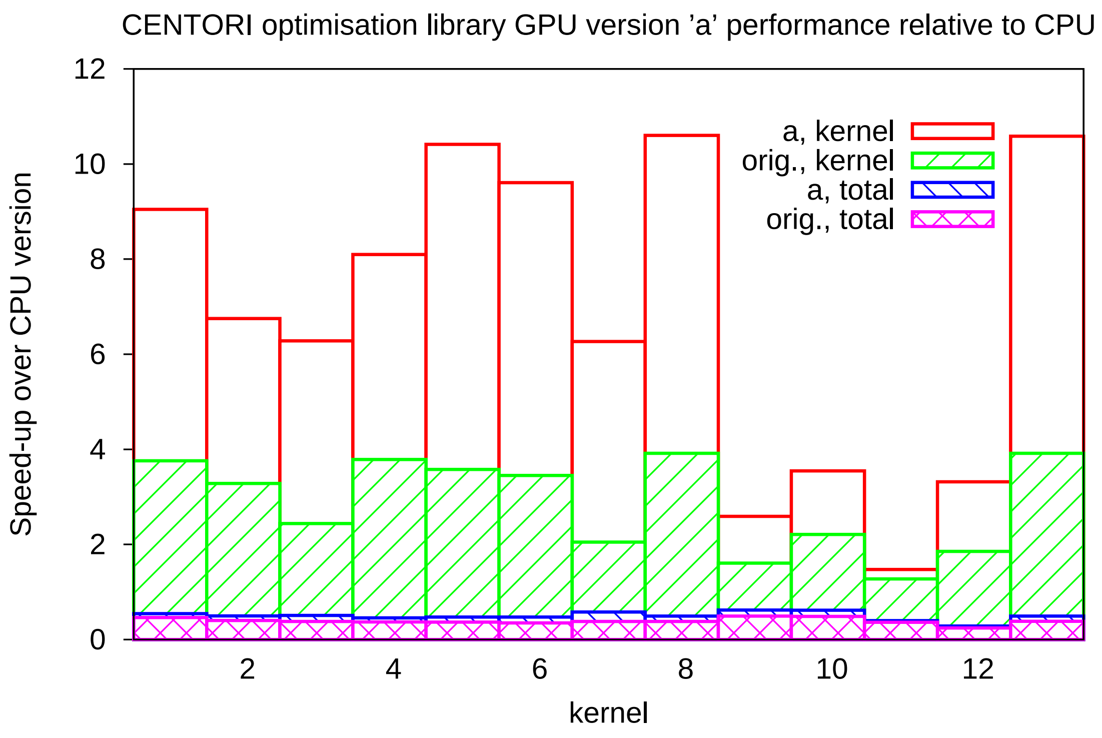


Figure 4.13: Speed-up with the 'a' optimisation (data restructured to improve coalescing). The results show the case where just the time to execute the kernel on the GPU is considered, and the case where the time for other operations such as copying data to a from the GPU is included. The time taken to restructure the data is not included.

It is important to note that the time required to restructure the data is not included in the timing measurements taken to make the graphs. This is because if this version of the GPU code were used with CENTORI, it might be possible to modify the CPU code so that the restructured array could be used throughout the application and therefore the restructuring process would not need to be performed every time a kernel is launched on the GPU. The sequential nature of CPUs means that in the CPU code the restructured array would probably suffer from poor performance as it is likely that the three vector components of each element would be accessed together, but they are no longer contiguous in memory. If it is not possible to rewrite the CPU code to resolve this, it might still be beneficial to use this approach as the improvement in performance of the kernels on the GPU might be greater than the decrease in performance of the CPU code. As an alternative, optimisation versions 'e' and 'f' (discussed below) present another method of obtaining coalesced memory access without restructuring the data on the CPU.

4.2.2 ‘b’: Eliminating unnecessary Global Memory accesses

The second optimisation, referred to as version ‘b’, further reduced Global Memory access in certain kernels by eliminating unnecessary reads. The same idea was used in the example (section 2.3.1) and in version 5 of the Ludwig GPU code. In some kernels data from Global Memory was used several times. As the GPU does not know whether this data has been modified by other threads between reads, it must load the data from Global Memory every time. By examining the code it was possible to determine that the data would not be modified by other threads, and so it was safe to prevent these unnecessary reads by loading the data into registers at the beginning of the kernel and then accessing the values stored in registers during the calculation. This optimisation was applied to suitable version ‘a’ kernels, as this optimisation could be used in addition to the restructuring of AT1 arrays.

An example of a kernel in which this optimisation was used is shown in Fig. 4.14. Kernel 1, which is used in the graph, contains three operations, each of which involves the same four double precision values. In version ‘b’ these were loaded into registers at the beginning of the kernel, eliminating eight unnecessary loads. Each of the operations also involves loading four other double precision values, which are different for each of the operations. This means that in total each kernel originally loaded twenty-four doubles, and in the optimised version this is reduced to sixteen, a saving of one third. It would therefore be expected that the performance of the kernel would be increased by approximately this amount. Surprisingly this optimisation actually resulted in a small decrease in performance for this kernel. The most obvious explanation for this would be that the increased use of registers limited the number of threads that could be simultaneously active, preventing the full computing power of the GPU from being utilised. The `nvcc` compiler reports, however, that this optimisation actually decreased register usage from 19 to 17. The cause of this behaviour is therefore unknown.

The version ‘b’ optimisation was also applied to other kernels, including kernel 4. In this case, a 22% reduction in Global Memory loads was obtained with just one double precision value stored in registers. This resulted in the speed-up increasing from about eight times to over ten times, as seen in Fig. 4.17.

4.2.3 ‘c’: Using Shared Memory to reduce register usage

In several kernels the same data was read by some or all of the threads in a thread block. This means that the threads will try to access the same location in Global Memory simultaneously. As Global Memory does not have a broadcast mechanism, these accesses will be serialised. Reading this data once per thread block and storing it in Shared Memory would therefore result in a significant reduction in Global Memory accesses. This was the optimisation implemented in version ‘c’ of the GPU code. One kernel suitable for this modification was kernel 1. This optimisation behaved as expected and produced a good improvement in performance, as can be seen in Fig. 4.14.

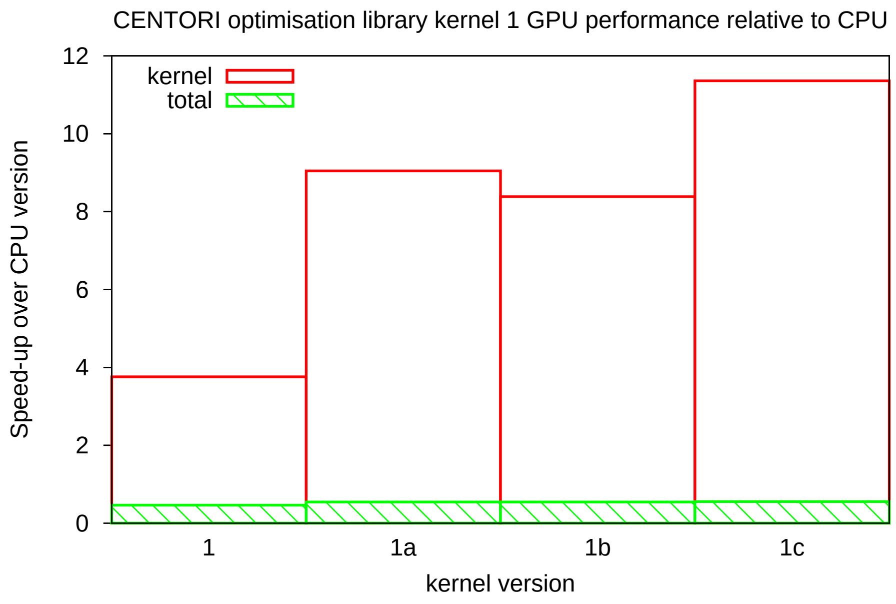


Figure 4.14: Speed-up of kernel 1 with various optimisations

A further optimisation that would potentially improve performance of this kernel would be to load the data stored in Shared Memory in version ‘c’ into Constant Memory. This would reduce the Global Memory accesses further to just one for each SM (whereas in the version ‘c’ optimisation it is one per thread block). Unfortunately Constant Memory can only be declared at file scope and with application lifetime, which means using it in this case would interfere with the other kernels.

As an experiment, the version ‘c’ optimisation was applied to kernel 7. This kernel is suitable for the version ‘b’ optimisation, since data in Global Memory is repeatedly, and unnecessarily, accessed in the kernel. This kernel differs from the other kernels to which the version ‘c’ optimisation was applied as the values stored in registers for each thread in version ‘b’ are different. The purpose of applying the optimisation to this kernel was to investigate whether using Shared Memory instead of registers was advantageous even if the data was not shared between threads. This meant that it was necessary to create an array in Shared Memory to store the values for each thread. The performance was unaffected by this modification, as can be seen in Fig. 4.15, confirming Nvidia’s claim that Shared Memory access is usually as fast as register access.

4.2.4 ‘d’: Processing the z dimension in stages

Communication with the author of the CENTORI optimisation library revealed that nz (the size of the problem domain in the z dimension) was usually 32 or less. This is suitable for GPUs as the grid of thread blocks is two-dimensional – it is not possible to stack thread blocks in the z dimension. The number of threads per block

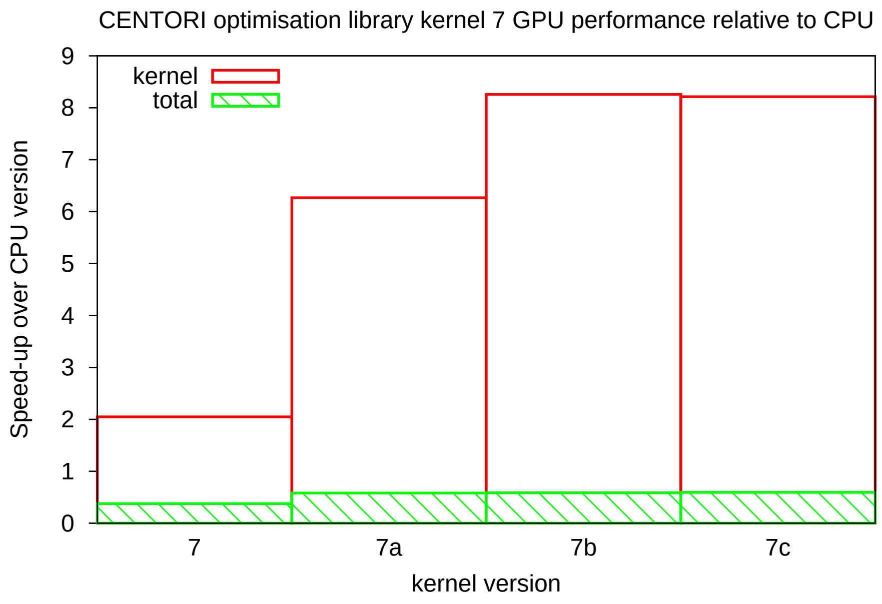


Figure 4.15: Speed-up of kernel 7 with various optimisations

in the z dimension is also limited to a maximum of 64. Alternative decomposition strategies are necessary for problem sizes with z greater than 64. These were not implemented for most of the kernels as it was felt that it would add unnecessary complexity to the code, and might reduce performance. It was decided to demonstrate that it was possible to do this, however, in version 'd' of kernel 8. The strategy used was to define the number of threads per block in the z dimension to be less than the maximum, and then to have each thread perform the operations in the kernel for as many elements in the z dimension as it took for the problem domain to be covered (i.e. $nz/$(number of threads per block in z dimension)). This was achieved by having a loop inside the kernel. Another advantage of this method is that it provides greater freedom to choose the number of threads per block in the z dimension. Previously this was fixed as nz, but now could be any allowable number. As the data in memory was contiguous in the x dimension, having more threads per block in this dimension would be likely to improve coalescing. The number of threads per block in the z dimension was therefore reduced to one, allowing the number in the x dimension to be increased to 128. The result, which can be seen in Fig. 4.16, reveals that this method can actually improve performance.

4.2.5 'e' and 'f': Alternative to restructuring

Kernel 4 was also used to demonstrate an alternative to restructuring AT1 arrays in the version 'a' optimisation. In version 'e' Shared Memory was used to achieve memory coalescing of AT1 arrays without first restructuring the data on the CPU. This was done by having all of the threads in a thread block load a contiguous block of data into Shared

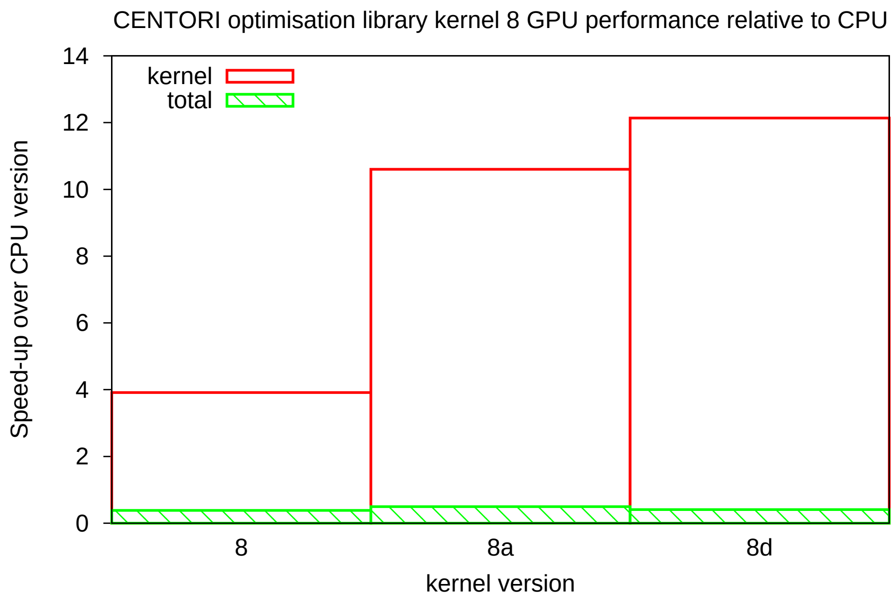


Figure 4.16: Speed-up of kernel 8 with various optimisations

Memory, starting from the first vector component of the first thread in the thread block. Most of the threads will be loading data that does not 'belong' to the element they will actually operate on. This load will be coalesced unless the number of threads per block is not a multiple of sixteen. When this is completed, one third of the data required by the thread block will be stored in Shared Memory. This is then repeated two more times, each time starting from the end point of the previous load. This results in all of the data being loaded using coalesced memory access without any restructuring. The threads can now perform the operations of the kernel, using the data in Shared Memory that corresponds to the element they are working on. Once the kernel operations are completed, if the array that stores the result was an AT1 array, then it must be copied from Shared Memory to Global Memory using a similar approach to the way in which the memory was loaded.

The kernel performance is slightly lower using this method compared to the restructuring version. If the time required to restructure the array is included (the 'incl. restructure' results on Fig. 4.17 – version 'e' does not require restructuring, so its equivalent is the normal 'total' result), then it is clear that using this method could improve the total runtime.

Another issue with this method is that the code becomes more complicated. An attempt to resolve this was made in version 'f', which uses separate functions to perform the Shared Memory loading and unloading. Device functions (functions on the GPU called by other functions running on the GPU) are usually inlined, and so it would be expected that the performance of this version would not be different from version 'e'. It appears that this was not the case, however, as the speed-up of version 'f' is slightly lower than that of 'e'. It is possible that the compiler was not able to understand the functions and

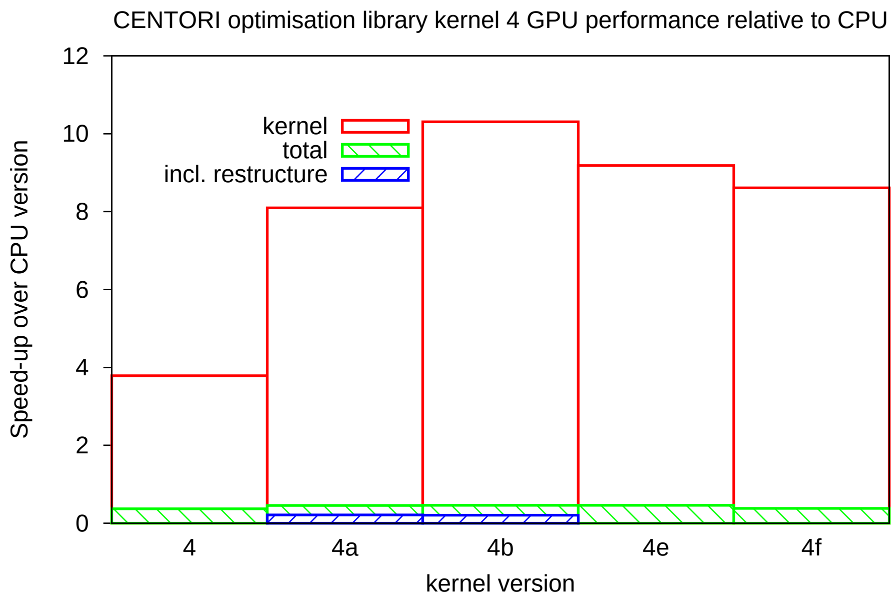


Figure 4.17: Speed-up of kernel 4 with various optimisations. In this chart, the 'total' measurement includes the cost of restructuring the data in the versions that perform this ('a' and 'b').

so decided not to inline them. It is clear that although the functioning of the code is not modified, the code in this version of the kernel is being treated differently by the compiler since the amount of Constant Memory used increased from 4 to 12 bytes.

4.2.6 'i': Eliminating unnecessary Global Memory reads to reduce impact of non-coalesced access

Kernel 11 was the poorest performing of the kernels when using the original GPU code. Version 'a' of the code made very little improvement, with the kernel only performing about 1.5 times faster on the GPU than the CPU. Significant effort was therefore expended attempting to improve the performance of this kernel. The kernel implements the gradient operator:

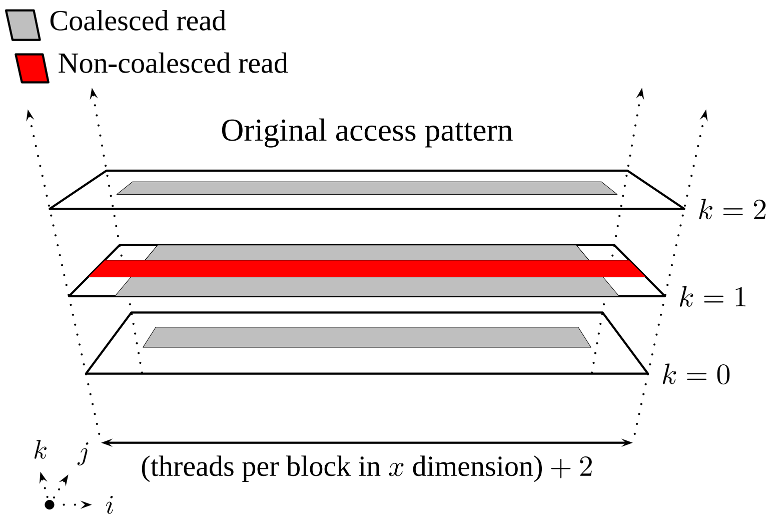
$$out_{i,j,k} = \left(\frac{a_{i+1,j,k} - a_{i-1,j,k}}{2dx}, \frac{a_{i,j+1,k} - a_{i,j-1,k}}{2dy}, \frac{a_{i,j,k+1} - a_{i,j,k-1}}{2dz} \right)$$

The problem with this operation is that achieving memory coalescing is difficult or impossible. This is because 'halo' elements are needed by each thread block: the first thread will need access to $a(i-1, j, k)$, and the last thread will similarly need access to data of the first element of the next thread block. Loading a single element from Global Memory is not possible, so this will result in significant wastage of memory bandwidth.

The idea pursued in attempting to resolve this issue was that if the loop over the z dimension described for kernel 8 version ‘d’ was used with only one thread per block in the z dimension, then in each loop iteration the threads in a thread block will operate on the elements ‘above’ those of the previous iteration (in the z dimension). This meant that much of the data loaded as $a(i, j, k + 1)$ in one iteration, would be used again the next iteration for $a(i + 1, j, k)$ and $a(i - 1, j, k)$, and then again in the iteration after that for $a(i, j, k - 1)$. This access pattern is visualised in Fig. 4.18. While the initial loading of this data would still be a non-coalesced access, storing this data could reduce the number of Global Memory loads in subsequent iterations (although the number of non-coalesced accesses would be the same). In the following discussion it will be assumed that there are n threads per block in the x dimension (where n is a multiple of 16), and one in the y and z dimensions. Before this optimisation was implemented, in each iteration every thread block would need to load the $n + 2$ elements $a(i, j, k), i = [a \times n - 1, \ldots, (a + 1) \times n + 1], j = b, k = c$ (for the ath thread block in the x dimension, the bth thread block in the y dimension, and during the cth iteration). This would be a non-coalesced access as the starting address is not a multiple of sixteen, and more than n elements are needed. Additional loads are also required for the n elements ‘above’ and ‘below’ in the z dimension, and n elements on each ‘side’ (in the y dimension). These loads would be coalesced. In the optimised version, only the two ‘side’ loads would be necessary, in addition to the load of the ‘above’ elements, although now $n + 2$ elements must be loaded in the ‘above’ case so it is no longer a coalesced access. The benefit of this optimisation would therefore be to reduce the Global Memory accesses from one non-coalesced load and four coalesced loads, to one non-coalesced load and two coalesced loads. While this does not solve the non-coalesced access, by reducing the total amount of data requested from Global Memory it was hoped to decrease the penalty of non-coalesced access. The cost of doing this is a much more complicated code, a requirement that there be only one thread per block in the z dimension, and most importantly a large usage of Shared Memory.

This optimisation was implemented gradually in versions ‘i’, ‘j’ and ‘k’. The differences between the versions included using registers more efficiently to store the result of expensive modulo operations rather than whether a thread was operating on the edge of the problem domain, which could be determined again when needed with a simple conditional statement based on the thread ID.

Despite the clear advantage of this method, Fig. 4.19 shows that no performance improvement was obtained. It is believed that this was caused by a number of issues. The main problem is that non-coalesced memory accesses were not reduced. The relatively fast coalesced accesses that were eliminated probably did not contribute much to the runtime as they were hidden by the much longer non-coalesced access. Another issue is the significantly increased register and Shared Memory usage. This limits the number of threads that can be active simultaneously, potentially resulting in much of the GPU’s SPs remaining idle during the calculation. Finally, the greatly increased complexity of the code means that each thread has to perform many more operations, reducing the time saved through decreasing memory loads.



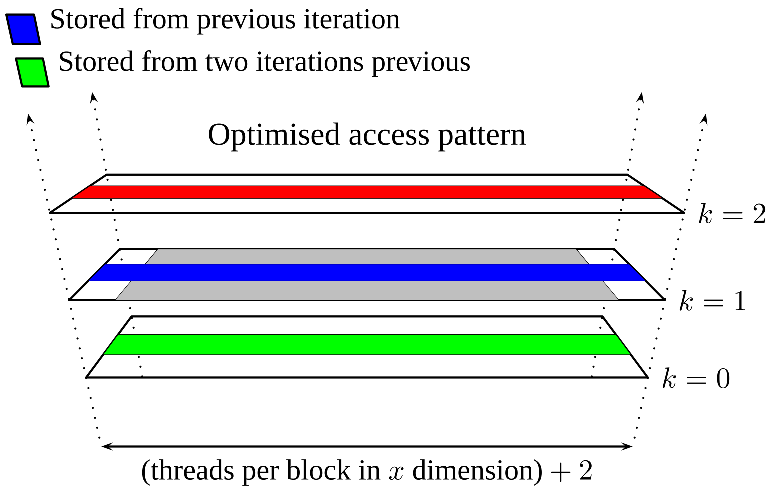

Stored from previous iteration
Stored from two iterations previous
Optimised access pattern
$k = 2$
$k = 1$
$k = 0$
(threads per block in x dimension) + 2

Figure 4.18: Diagrams showing the change in the data access pattern between the original implementation (top) and the version i, j, and k optimisation (bottom). The shaded regions represent elements that are read from Global Memory, or are stored in Shared Memory, during an iteration (at least the third iteration).

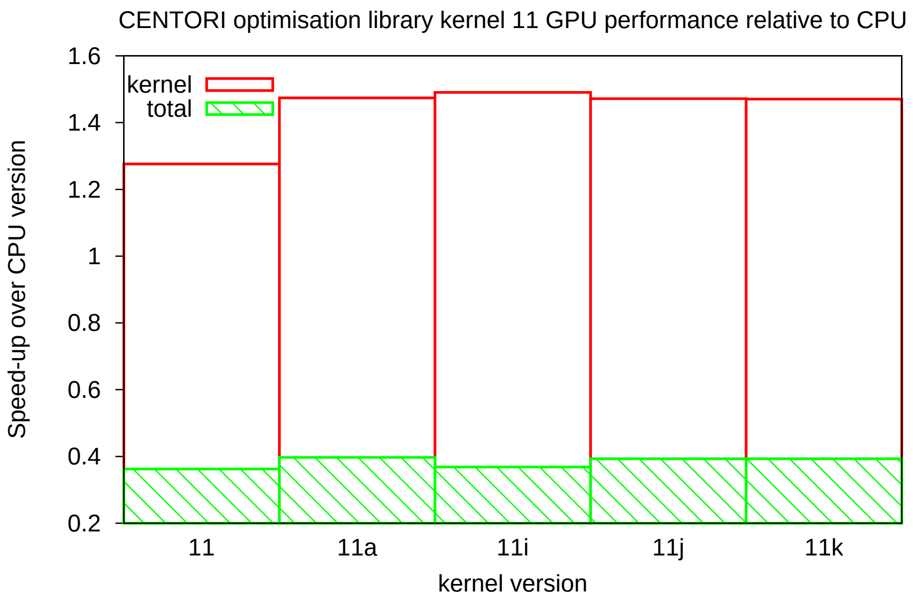


Figure 4.19: Speed-up of kernel 11 with various optimisations

4.2.7 ‘x’: Using the CUBLAS library

The CUBLAS library mentioned in section 2.3 contains functions that perform the same operations as some of the kernels in the CENTORI optimisation library. One such operation is the multiplication of a matrix by a scalar, which is the purpose of kernel 12. As the same scalar is multiplied by each of the vector components for every element in the AT1 array, the version ‘b’ optimisation could be used to reduce unnecessary memory loads as previously described. The version ‘c’ optimisation of using Shared Memory instead of registers for storing this value was also applied. The performance improvements obtained through these optimisations are obvious from Fig. 4.20. In the graph, it can be seen that the performance is further improved in version ‘x’. This is an implementation of the kernel that uses the cublasDscal matrix-scalar multiplication operation in CUBLAS. Using this operation was very easy – it does not require a separate ‘kernel’ function as CUBLAS calls are made from host code, and the performance is better than the other purpose-written versions.

The CUBLAS library should be used with caution, however, as it is not always suitable for the intended operation. An example of this can be seen in the case of kernel 6. This kernel implements a dot product operation. The CUBLAS library includes a cublasDdot operation, which appears to perform the desired operation. The kernel computes the dot product of the three vector components of each array element, however, while the CUBLAS operation computes the dot product of every element in the vector passed to it. It is therefore necessary to loop over all of the matrix elements on the host, calling the CUBLAS library for the vector associated with each element. This results in extremely poor performance, as each element’s vector must be individually copied from the CPU

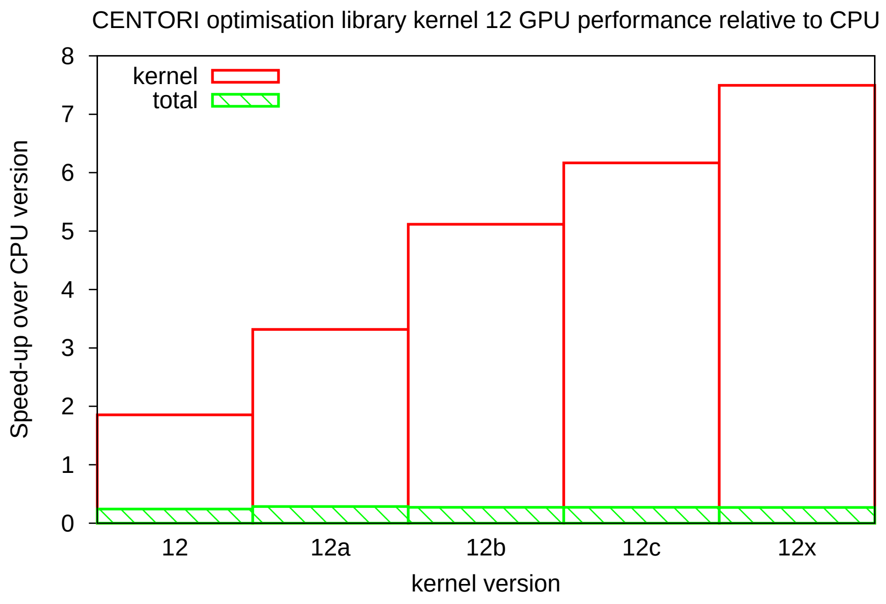


Figure 4.20: Speed-up of kernel 12 with various optimisations

to the GPU, the dot product computed, and the result copied back to the CPU. Fig 4.21 shows the performance of the different versions of the kernel, rescaled so that the performance of version ‘x’ can be seen. Note that since the total runtime of all of the GPU functions in version ‘x’ is practically equivalent to the kernel execution time, the ‘total’ and ‘kernel’ results are the same.

4.2.8 Final results

The final results for CENTORI, the fastest version of each kernel, are shown in Fig. 4.22. It is clear from the excellent performance improvements obtained by many of the kernels that the type of operations performed by CENTORI are very suitable for GPU acceleration. It is also clear that for the GPU implementation to be useful, these operations must be combined so that data transfer between the GPU and CPU only has to be performed once for several kernels. The scaling with problem size (Fig. 4.23) shows that small problem sizes cause low utilisation of the GPU resulting in poor speed-up. A graph showing the speed-up for varying problem size in the z dimension is available in Appendix B.2.

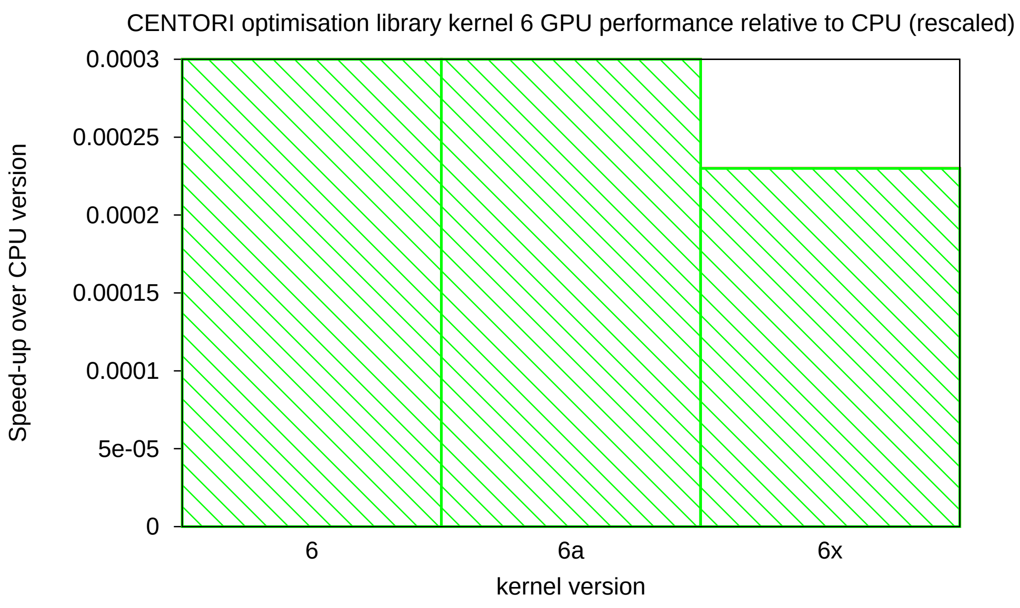


Figure 4.21: Speed-up of kernel 6 with various optimisations. The y-axis has been rescaled so that the speed-up of version ‘x’ is visible.

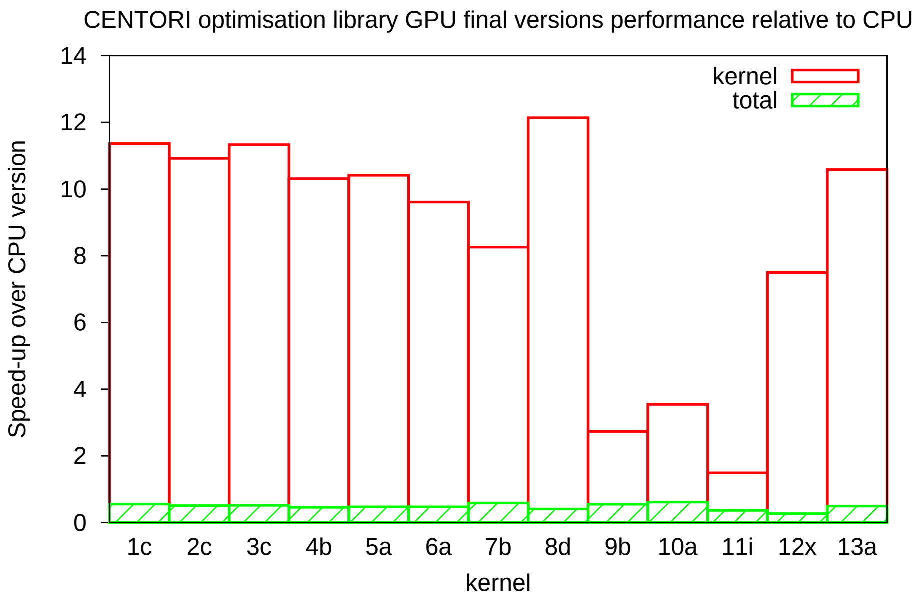


Figure 4.22: Speed-up of all kernels when using the best-performing optimisations

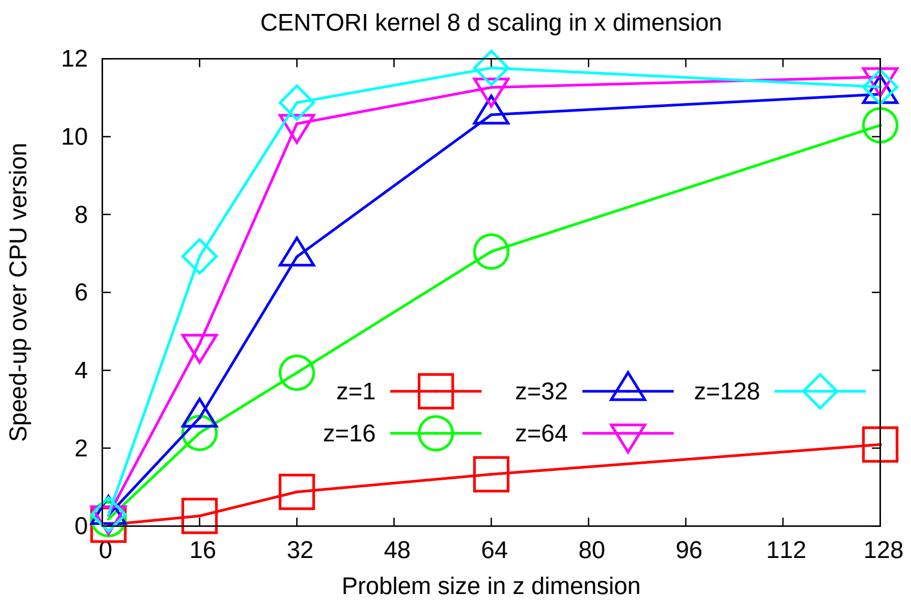


Figure 4.23: Speed-up of kernel 8 version ‘d’ for varying problem size in the x dimension, shown for various problem sizes in the z dimension. The problem size in the y dimension was 128 in all cases.

Chapter 5

Testing

Testing is a very important task whenever a code is modified. Making an application run faster is of no value unless the results it produces are known to still be correct. Since porting a code to run on a GPU requires very large alterations to be made, the risk of errors being introduced is high, and so it is essential that the code is thoroughly tested.

For all of the codes that were ported, the testing methodology used was similar: the output of the code that was run on the GPU was compared to the output of that section of code from the original, unaltered CPU version. Differences between the codes meant that how this was implemented varied for each application.

One issue encountered during testing was that the results produced by the code executed on the GPU was not always identical to the CPU version. The most probable reason for this is that different compilers were used to compile the two versions. Code that runs on the GPU is compiled using a special compiler developed by Nvidia and based on the Open64 compiler, whereas GNU Compiler Collection (GCC) compilers were used for the CPU code. This means that the optimisations applied to the code will not be the same, and so the order in which calculations are performed will most likely be different. The non-commutativity of floating point arithmetic would then result in different outputs being produced. Another possible cause of the discrepancies is that the calculations are being performed on different architectures. Even when the same floating point operation is performed on two different CPUs, the results are often different, as discussed in [21]. GPU architectures were not designed to perform non-graphics calculations, and especially not double precision arithmetic. While the CUDA language successfully hides most of the underlying details of how GPGPU is implemented on a GPU, it is probable that this emulation is achieved through the use of a variety of techniques. The result would be that on a GPU architecture, calculations may be performed in a very different manner to how they are done on a CPU. One way in which this is exposed is by GPUs not strictly conforming to the IEEE 754 standard in all areas. It is therefore necessary to verify that results from the GPU code are within an acceptable margin of error of the original CPU results, rather than identical. As double precision representation can store about 16 significant figures, the results should be approximately equal up to this level. How close is considered to be ‘acceptable’ depends on the requirements of each application.

The issue of accuracy is further complicated by the varying precision of floating point numbers. This is defined in detail by [15], but one feature is that the precision with which numbers can be stored decreases as the numbers get larger. This means that an absolute value cannot be used for the maximum difference between the correct results and the results under test (for example, saying that they must be equal up to ten decimal places) if the results have a large range. One method that could be used in such circumstances is to check the percentage error rather than the absolute error. For this dissertation, a maximum percentage error was used that was quite close to the maximum accuracy possible with double precision, and that also caused the test to pass for GPU implementations which were believed to be correct. A less computationally expensive method, but which is more complicated to implement, involves specifying the maximum acceptable difference in terms of the number of representable floating point numbers. For small numbers, the distance between representable floating point numbers is small, but it can be quite large for bigger numbers. This is possible because of the way in which the bit representation of double precision numbers is defined. Reading the value of the double precision values as an integer type that is of the same length (`long long` on the machine on which the test was performed) results in the integer with the same bit representation being obtained. Two consecutive floating point numbers differ in the last element of their bit representation as the mantissa is stored at the end of the pattern. This conveniently mirrors the situation of integers. Subtracting the integer representations of the double precision results produced by the CPU and GPU versions therefore allows the number of representable double precision number between them to be computed. A more in-depth discussion of this technique for single precision numbers is available at [5].

To facilitate testing, preprocessor macros were used for each application to specify whether testing should be enabled or not (`GPU_TEST` enables testing), and whether the CPU or GPU versions of the code should be executed (`GPU` enables the GPU version, otherwise the CPU is used). In the applications where testing was performed by writing the results to disk for comparison, the testing macro was enabled and the code executed on the CPU and GPU in turn. While this approach requires recompilation of the code, the alternative, passing arguments at runtime and using conditional statements in the code based on these to choose whether testing is performed, would interfere with timing measurements.

5.1 Ludwig

Ludwig contains some random elements, which could have made testing more difficult, however the output of the section of code ported (arrays `site` and `u`) is not affected by this, and so is always the same. The strategy used, therefore, was to write to disk the data that was altered by the GPU. The output was written using the exponential notation format so that numbers could be printed with a specified number of significant figures (currently defined to be 15). The iteration during which the output is written is specified using the NUMOFITS preprocessor macro. When the code is run in parallel using MPI, each process appends its rank to the filenames of this output data. The CPU and GPU versions must be run with the same problem domain dimensions and number of processes so that these files can be compared. A small script was written using the Python language to perform the comparison by computing the percentage difference between corresponding values in the files saved to disk. A required precision of $1 \times 10^{-7}\%$ was used. For large problem sizes this comparison can be quite slow, so a quick verification can be made by visually inspecting the results to check that they are approximately equal. This required precision needed to be less accurate for Ludwig than for the other codes (see below) in order for the tests to pass. The magnitude of the errors appeared to decrease as the number of iterations increaseed. The reason for this is unclear. Compiling the CPU code with different rounding modes and examining the variation in the results may indicate whether the numerical stability of the algorithm is responsible. An sample of the output data is available in Appendix D.1

5.2 FAMOUS

As a single run of FAMOUS takes several hours, running the entire code every time the code needed to be tested was impractical. This was resolved by executing the original CPU version and capturing the input to and output from the section of code that was ported. These were written to files. A framework was then written to load the input data, execute either the CPU or GPU version of the section of code that was ported (chosen by a preprocessor macro, as previously described), and write the output to a file so that it could be compared to the output from the original version.

As with Ludwig, the output was written using exponential notation (to 13 significant figures) and compared by computing the percentage difference ($1 \times 10^{-13}\%$ was used). A sample of this data is presented in Appendix D.2.

5.3 CENTORI Optimisation Library

The structure of this code was different from the others due to it being a framework for timing and testing many small, independent kernels, rather than a function that is part of a larger code. This meant that a slightly different testing strategy to that used in the other applications was more appropriate.

When testing is enabled by the preprocessor macro, the CPU and GPU versions of the selected kernels are each run once with the same input, and the outputs compared.

Two functions to perform the comparison are supplied. The first implements a percentage error check with a required precision of $1 \times 10^{-12}\%$. The second uses the elegant method mentioned above of calculating the number of double precision floating point numbers between the two versions of each element of the output data. It then checks that this is below the specified maximum (currently 10).

Chapter 6

Conclusion

This dissertation investigated the GPU acceleration of five HPC applications: the CENTORI optimisation library, FAMOUS, Ludwig, CHILD, and CSMP. The first three of these were successfully ported.

Good performance was achieved for Ludwig and the CENTORI code. Type 1 scaling (performance variation with problem size) was good for Ludwig, with the speed-up in kernel execution being approximately constant at 6.5 times. The total time (kernel execution and memory transfer) was constant at 2.9 times faster, except for the case when the problem size in the z dimension was one, which gave a total speed-up of 1.6 times. Type 2 scaling (fixed problem size, varying number of GPUs) performance was stable at 6.6 times faster from one to eight GPUs, but then declined to 5.5 times for 32 GPUs. Speed-ups for ten of the thirteen kernels in CENTORI were between 8 and 12 times. The remaining three kernels had speed-ups between 1.5 and 3.5 times.

GPUs are only suitable for certain types of codes, primarily those that are similar to the graphics-like calculations that GPUs were designed to perform. This disqualifies some existing applications, such as CHILD, which are not capable of being decomposed into many SIMD-like threads. It is possible that for some of these cases, a large restructuring of the code may make it more conducive to massive parallelism.

Despite the efforts of companies such as Nvidia to reduce the complexity of GPU software development by providing a simple programming language and many useful tools, several difficulties were encountered. One of these was the requirement that all of the memory needed by the code that executes on the GPU be explicitly transferred onto the GPU. Additionally any other function that is called by the code must also be ported as CPU functions cannot be accessed by the GPU. Some elements of the design of Ludwig, such as a large use of file scope identifiers, made this process time-consuming. Due to unusual compiler linking behaviour, it was necessary to place a lot of code together in single files, when it would have been more desirable for it to be split into separate files. Another issue that caused difficulty in the initial porting of the codes was the necessity that they be written in C-like CUDA to be compiled for the GPU. For the Fortran codes FAMOUS and CENTORI this meant that part of the code also needed to be converted from Fortran to C. This issue should be improved with the upcoming Fortran CUDA compiler. As was discovered during this dissertation, debugging can also be problematic on a GPU. The recent CUDA-GDB debugger goes some way to rectifying this situation, but the inability to perform some of the standard debugging methods, such as printing-out values at specified points using a simple `printf` statement from the GPU, often makes this more challenging than on a CPU. The main problem that was encountered with the use of GPUs, however, was that even if large efforts are expended to obtain a significant speed-up in the processing of the code that executes on the GPU, the time taken to transfer memory between the CPU and GPU can greatly reduce these gains.

As the objective of using a GPU is to accelerate a code, obtaining good performance is critical. Simply getting a code to execute on a GPU will almost certainly not result in the dramatic speed-up that is expected. Some optimisations, such as reducing unnecessary Global Memory loads by using registers can provide a good performance improvement with a small amount of work. Others, such as restructuring data to improve memory access coalescing, may be a large undertaking, possibly requiring changes to the rest of the CPU code as well. Both of these strategies were used during the optimisation of Ludwig and CENTORI along with others including the use of Constant and Shared Memory. With so many conflicting factors to consider – maximising the amount of threads that can be simultaneously active by limiting usage of registers and Shared Memory, while also reducing Global Memory accesses, making sure that any Global Memory accesses that are necessary are coalesced, avoiding conditional statements that lead to divergent warps, attempting to ensure that Shared Memory accesses do not cause bank conflicts, and many other issues, finding a compromise that balances all of these was probably the most difficult aspect of GPU development.

Further work was suggested, including porting more of Ludwig to run on a GPU to reduce the amount of memory that must be transferred between the CPU and GPU each iteration. If this was performed, and the kernels in CENTORI were combined so that a memory transfer would not be required for each kernel launch, then it is likely that the amount of runtime spent on operations other than kernel execution could be reduced, resulting in speed-ups close to those of the kernel execution.

The enormous interest that has been displayed in the potential of accelerating codes, primarily HPC applications, by the use of GPUs suggests that this is a field that will continue to develop rapidly for the foreseeable future.

Appendix A

GPU memory functions

Timing results for various GPU memory functions are presented in Fig. A.1. The memory transfer time increases at a rate of about 1ms per MB. This reveals a memory transfer speed of almost 1000MB/s, which is less than the theoretical peak transfer speed. Using pinned memory (also referred to as page-locked memory) resulted in a more than doubling of the transfer speed (not shown on graph). Page-locked memory may result in decreased performance of the host system, however. A close-up of the data for very small data sizes (inset of Fig. A.1) shows that a different mechanism appears to be used which allows the allocation and freeing of memory with greatly reduced overhead.

This mechanism is not just for the first 2-4KB of allocated memory, however, as can be seen from Fig. A.2. In this chart, 64 byte blocks of memory are allocated until a total of 8KB has been allocated on the device. The time per memory allocation does not jump when the total is between 2 and 4 KB, which shows that whether the allocation is made in the shorter time is based on the size of memory allocated rather than the total that has been allocated before the allocation is performed. The time per allocation increases with the total amount of memory that has been allocated. This behaviour is expected as it will take longer for a suitable area in the memory to be located for the allocation when there is less free memory available.

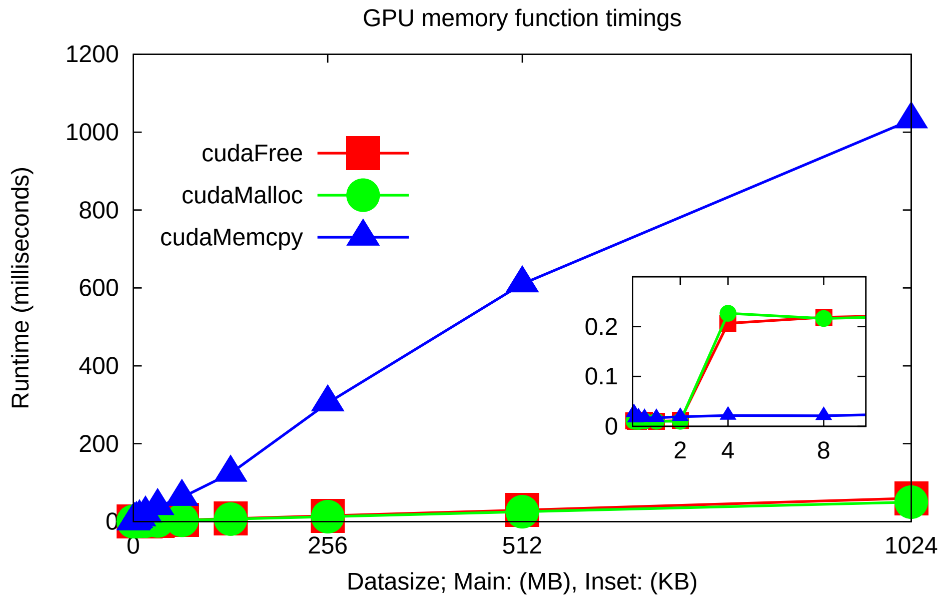


Figure A.1: The time required to allocate, free, and copy to from the host, memory on a Tesla C870 GPU

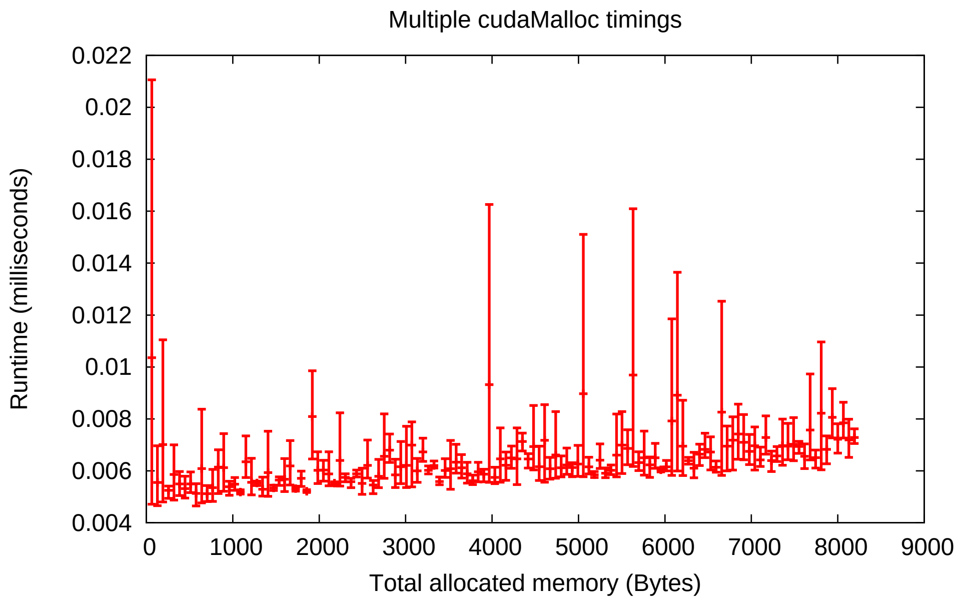


Figure A.2: The time required to repeatedly allocate 64 bytes of memory on a Tesla C870 GPU

Appendix B

Additional optimisation charts

B.1 Ludwig GPU v1

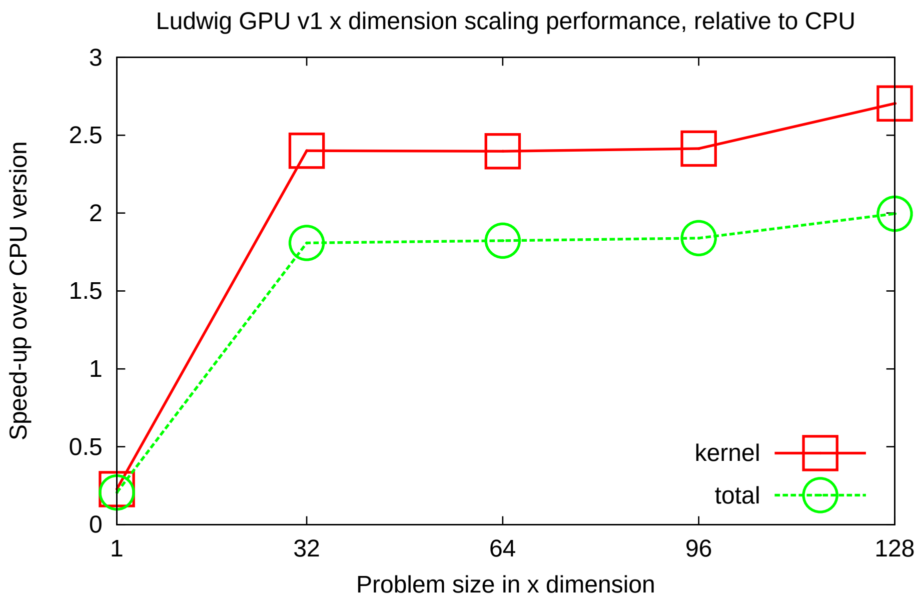


Figure B.1: Speed-up obtained by version 1 of the GPU implementation of a section of Ludwig for varying the problem size in the x-dimension (with problem size in the y and z dimensions fixed at 128) on 1 GPU (compared to 1 CPU core). Data is shown for the case when only the execution of the kernel is timed, and the case when other GPU-related operations, such as copying data to and from the GPU, are included.

B.2 CENTORI optimisation library scaling

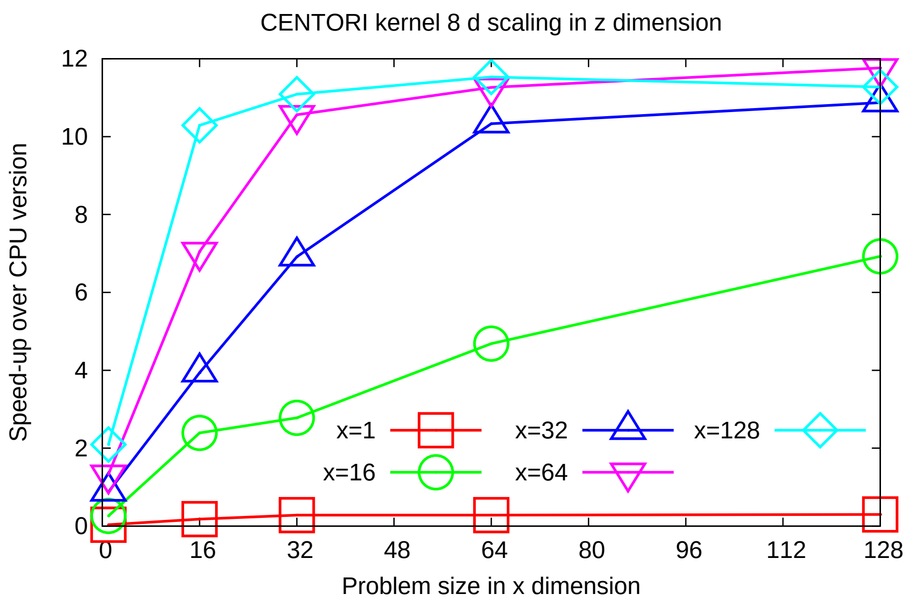


Figure B.2: Speed-up of kernel 8 version ‘d’ for varying problem size in the z dimension, shown for various problem sizes in the x dimension. The problem size in the y dimension was 128 in all cases.

Appendix C

Profiles of considered applications

The twenty functions that consume the most runtime according to profiles created using `gprof` are presented for Ludwig, FAMOUS, and CHILD.

C.1 Ludwig

% time	calls	total Ks/call	name
79.27	10	1.15	MODEL_collide_binary_lb
4.40	5242880	0.00	f_grad_phi
4.33	10	0.06	propagation
3.69	5242880	0.00	f_delsq_phi
1.21	5767168	0.00	get_phi_at_site
1.07	1	0.15	TEST_momentum
0.85	5242880	0.00	fe_chemical_stress_symmetric
0.57	22544384	0.00	site_map_get_status
0.36	8	0.01	get_N_offset
0.36	47193896	0.00	get_site_index
0.36	10485760	0.00	phi_get_delsq_phi_site
0.36	5242880	0.00	fe_chemical_potential_brazovskii
0.36	524288	0.00	set_phi
0.36	1	0.08	TEST_statistics
0.28	524288	0.00	set_rho
0.21	5242880	0.00	hydrodynamics_get_force_local
0.21	5242880	0.00	hydrodynamics_set_velocity
0.21	11	0.02	phi_compute_phi_site
0.14	16777216	0.00	phi_get_phi_site
0.14	5242880	0.00	phi_get_grad_phi_site
0.14	10	0.00	free_energy_is_brazovskii

Table C.1: Partial profile of Ludwig

C.2 FAMOUS

%		total	
time	calls	Ks/call	name
9.40	76246689	0.00	trans_source_coeff_
7.56	207360	0.00	solver_triple_
5.95	3533760	0.00	two_coeff_region_
5.24	17668800	0.00	single_scattering_
4.61	17668800	0.00	rescale_tau_omega_
4.13	875520	0.00	grey_extinction_
3.19	76246689	0.00	two_coeff_
2.94	4487040	0.00	scale_absorb_
2.73	3326400	0.00	solver_triple_app_scat_
2.26	875520	0.00	solve_band_k_eqv_
1.92	3533760	0.00	solver_homogen_direct_
1.74	25920	0.00	uv_adj_
1.47	16446970	0.00	qsat_
1.44	25920	0.00	convect_
1.38	51120	0.00	ai_calc_
1.23	102240	0.00	isoflux_
1.23	4377600	0.00	rescale_asymmetry_
1.22	51840	0.00	tracer_
1.14	5444892	0.00	solar_coefficient_basic_
1.02	3055776	0.00	staccum_
0.90	207360	0.00	triple_solar_source_

Table C.2: Partial profile of FAMOUS

C.3 CHILD

% time	calls	total Ks/call	name
12.14	746059284	0.00	tLNode::AddDrArea(double)
8.37	808757473	0.00	tLNode::getDownstrmNbr()
7.78	9606401	0.00	tStreamNet::RouteFlowArea(tLNode*, double)
7.01	851952172	0.00	tEdge::getDestinationPtrNC()
6.13	918235613	0.00	tNode::getBoundaryFlag() const
3.68	833777226	0.00	tLNode::getFloodStatus() const
3.56	144156836	0.00	tListIter::Next()
2.54	144149222	0.00	tMeshListIter::IsActive() const
1.85	144149223	0.00	tNode::isNonBoundary() const
1.84	129474683	0.00	tEdge::getLength() const
1.69	326158	0.00	tEdge::getOriginPtrNC()
1.69	4001	0.67	tStreamNet::CalcSlopes()
1.64	18831868	0.00	tLNode::calcSlope()
1.58			tLNode::getZOld() const
1.46	9215863	0.00	tLNode::setTau(double)
1.44	61546254	0.00	tListIter::Next()
1.29	9604000	0.00	tLNode::EroDep(int, tArray, double)
1.23	65759487	0.00	tList::getIthDataRef(int) const
1.15	4001	0.32	tStreamNet::FlowDirs()
1.15	4000	0.39	tStreamNet::FindChanGeom()
1.06	144156836	0.00	tListIter::NextP()

Table C.3: Partial profile of CHILD

Appendix D

Output of CPU and GPU versions

D.1 Ludwig

CPU $site[\,].g[\,]$	GPU $site[\,].g[\,]$
1.451018433467763e-02	1.451018433467763e-02
4.36283165719353e-05	4.36283165719353e-05
4.39637627898369e-05	4.39637627898370e-05
-9.46132536959975e-07	-9.46132536960009e-07
4.27271601743938e-05	4.27271601743937e-05
4.30626064002499e-05	4.30626064002499e-05
4.47196841291997e-05	4.47196841291996e-05
5.65710179982962e-07	5.65710179982873e-07
4.34830815519740e-05	4.34830815519740e-05
1.23660262034103e-06	1.23660262034105e-06
-1.23660261465461e-06	-1.23660261465464e-06
4.41539740002865e-05	4.41539740002866e-05
-5.65710158387593e-07	-5.65710158387498e-07
4.29173713425166e-05	4.29173713425166e-05
4.45744490927542e-05	4.45744490927542e-05
4.49098953152107e-05	4.49098953152107e-05
9.46132509678182e-07	9.46132509678222e-07
4.36732926956582e-05	4.36732926956582e-05
4.40087389260691e-05	4.40087389260692e-05
8.16671012028223e-03	8.16671012028222e-03

Table D.1: A sample of ‘siteout2.dat’ and ‘siteout2_gpu.dat’, the output produced by the process with MPI rank 2 of the CPU and GPU versions of Ludwig respectively, when testing is enabled. These contain the values of $site[\,].f[\,]$ and $site[\,].g[\,]$ to 15 significant figures, of which only the latter is shown here.

D.2 FAMOUS

CPU	GPU
0.9008273612708E+00	0.9008273612708E+00
0.9008334868459E+00	0.9008334868459E+00
0.9008426431994E+00	0.9008426431994E+00
0.9048112450492E+00	0.9048112450492E+00
0.9048028763758E+00	0.9048028763758E+00
0.9047976360945E+00	0.9047976360945E+00
0.9048014548452E+00	0.9048014548452E+00
0.9047993041983E+00	0.9047993041983E+00
0.9047904469325E+00	0.9047904469325E+00
0.9047849018919E+00	0.9047849018919E+00
0.9047801464729E+00	0.9047801464729E+00
0.9038258426993E+00	0.9038258426993E+00
0.9038311430273E+00	0.9038311430273E+00
0.9038367565481E+00	0.9038367565481E+00
0.9038458506984E+00	0.9038458506984E+00
0.9037912881280E+00	0.9037912881280E+00
0.9037852262404E+00	0.9037852262404E+00
0.9037875463351E+00	0.9037875463351E+00
0.9047306296501E+00	0.9047306296501E+00
0.9047435047256E+00	0.9047435047256E+00

Table D.2: A sample of ‘trans_source_coeff_out_cpu.dat’ and ‘trans_source_coeff_out_gpu.dat’, the output produced by the CPU and GPU versions of FAMOUS respectively, when testing is enabled. This shows the values of the data modified by the ported section of code to 13 significant figures.

Appendix E

Modification to workplan

E.1 Double precision instead of single precision

During the preparation stage for the dissertation, it was anticipated that single precision would be used for the GPU implementation of the ported codes. This was because it was believed that access to only one GPU system capable of performing double precision would be available (at Daresbury Laboratory). It was known that a double precision-capable GPU was being installed at the Edinburgh Computing and Data Facility (ECDF), but it was not guaranteed that this would be ready early enough in the dissertation period for it to be useful. One of the most important risk mitigation strategies used was to have access to several GPU systems on which the dissertation work could be performed. It was therefore decided that it would not be possible to use double precision.

Shortly before the start of the dissertation research, however, the ECDF GPU was ready for use. There was therefore no longer any additional risk from using double precision. Additionally, the results produced if double precision were used would be more useful as the original CPU codes used double precision, and so the research would be more directly applicable. Double precision was therefore used throughout the dissertation research, except for the performance comparison of the single and double precision versions of Ludwig GPU v6 in section 4.1.6.

Bibliography

[1] ATI Stream Software Development Kit. `http://developer.amd.com/gpu/ATIStreamSDK`.

[2] K. Barros, R. Babich, R. Brower, M.A. Clark, and C. Rebbi. Blasting through lattice calculations using CUDA. *Arxiv preprint arXiv:0810.5365*, 2008.

[3] A. Buttari, J. Dongarra, J. Langou, J. Langou, P. Luszczek, and J. Kurzak. Mixed precision iterative refinement techniques for the solution of dense linear systems. *International Journal of High Performance Computing Applications*, 21(4):457, 2007.

[4] Community Surface Dynamics Modeling System: CHILD. `http://csdms.colorado.edu/wiki/Model:CHILD`.

[5] Comparing floating point numbers. `http://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm`.

[6] CSMP++ Wiki. /urlhttp://csmp.ese.imperial.ac.uk/wiki/Home.

[7] CUDA-enabled products. `http://www.nvidia.com/object/cuda_learn_products.html`.

[8] CUDA Zone. `http://www.nvidia.com/object/cuda_home.html`.

[9] CUDPP: CUDA Data Parallel Primitives. `http://gpgpu.org/developer/cudpp`.

[10] J.-C. Desplat, I. Pagonabarraga, and P. Bladon. Ludwig: A parallel lattice-boltzmann code for complex fluids. *Computer Physics Communications*, 134(3):273 – 290, 2001.

[11] G. I. Egri, Z. Fodor, C. Hoelbling, S. D. Katz, D. Nógrádi, and K. K. Szabó. Lattice qcd as a video game. *Computer Physics Communications*, 177(8):631 – 639, 2007.

[12] Flagon. `http://flagon.wiki.sourceforge.net/`.

[13] E. B. Ford. Parallel algorithm for solving kepler's equation on graphics processing units: Application to analysis of doppler exoplanet searches. *New Astronomy*, 14(4):406 – 412, 2009.

[14] A. Gray, M. Ashworth, J. Hein, F. Reid, M. Weiland, T. Edwards, R. Johnstone, and P. Knight. A Performance Comparison of HPCx and HECToR. Technical report, HPCx, 2008.

[15] Ieee standard for binary floating-point arithmetic. *ANSI/IEEE Std 754-1985*, pages –, Aug 1985.

[16] C. Jones. Constructing a FAMOUS ocean model. Technical report, Hadley Centre, 2002.

[17] P. J. Knight, C. M. Roach, A. Thyagaraja, D. J. Applegate, and N. Joiner. Arithmetising plasma turbulence: the 'final frontier' of classical physics. *HPCx News*, pages 4 – 6, Autumn 2005.

[18] D. Komatitsch, D. Michéa, and G. Erlebacher. Porting a high-order finite-element earthquake modeling application to nvidia graphics cards using cuda. *Journal of Parallel and Distributed Computing*, 69(5):451 – 460, 2009.

[19] Stanford University Graphics Lab. BrookGPU. `http://www-graphics.stanford.edu/projects/brookgpu/`.

[20] S. K. Matthäi, S. Geiger, and S. G. Roberts. *The Complex Systems Platform CSP5.0: User's Guide*. ETH, 5th edition, 2004.

[21] D. Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3):1–41, 2008.

[22] Nvidia Corporation. *CUBLAS: CUDA BLAS*, 2.1 edition.

[23] Nvidia Corporation. *CUDA Reference Manual*, 2.2 edition.

[24] Nvidia Corporation. *CUFFT: CUDA FFT*, 2.1 edition.

[25] Nvidia Corporation. *Nvidia CUDA Programming Guide*, 2.2 edition.

[26] Nvidia Tesla S1070. `http://www.nvidia.co.uk/object/tesla_s1070_uk.html`.

[27] OpenCL. `http://www.khronos.org/opencl/`.

[28] Portland Group Accelerator compilers. `http://www.pgroup.com/resources/accel.htm`.

[29] Portland Group CUDA Fortran compiler. `http://www.pgroup.com/resources/cudafortran.htm`.

[30] T. Preis, P. Virnau, W. Paul, and J. J. Schneider. Gpu accelerated monte carlo simulation of the 2d and 3d ising model. *Journal of Computational Physics*, 228(12):4468 – 4477, 2009.

[31] RapidMind Multi-core Development Platform. `http://www.rapidmind.com`.

[32] A. Richardson and A. Gray. Utilisation of the GPU architecture for HPC. Technical report, HPCx, 2008.

[33] R. S. Smith, J. M. Gregory, and A. Osprey. A description of the famous (version xdbua) climate model and control run. *Geoscientific Model Development*, 1(1):53–68, 2008.

[34] S. Tomov, M. McGuigan, R. Bennett, G. Smith, and J. Spiletic. Benchmarking and implementation of probability-based simulations on programmable graphics cards. *Computers & Graphics*, 29(1):71–80, 2005.

[35] G. E. Tucker, S. T. Lancaster, N. M. Gasparini, and R. L. Bras. The Channel-Hillslope Integrated Landscape Development (CHILD) Model. In R.S. Harmon and W.W. Doe, editors, *Landscape erosion and evolution modeling*, pages 349 – 388. Plenum Pub Corp, 2001.

[36] G. E. Tucker, S. T. Lancaster, N. M. Gasparini, R. L. Bras, and S. M. Rybarczyk. An object-oriented framework for distributed hydrologic and geomorphic modeling using triangulated irregular networks. *Computers & Geosciences*, 27(8):959 – 973, 2001.

[37] S.Z. Ueng, M. Lathara, S.S. Baghsorkhi, and W.W. Hwu. CUDA-lite: Reducing GPU programming complexity. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*. Springer, 2008.

[38] Stuart D.C. Walsh, Martin O. Saar, Peter Bailey, and David J. Lilja. Accelerating geo-science and engineering system simulations on graphics hardware. *Computers & Geosciences*, In Press, Accepted Manuscript:–, 2009.

[39] Y. Zhao. Lattice Boltzmann based PDE solver on the GPU. *The Visual Computer*, 24(5):323–333, 2008.