Utilisation of the GPU architecture for HPC

A. Richardson, A. Gray
EPCC, The University of Edinburgh, James Clerk Maxwell Building, Mayfield Road, Edinburgh, EH9 3JZ, UK

December 15, 2008

Abstract

We describe the GPU architecture and discuss the potential for utilisation of this by HPC applications. We describe the porting of an HPC benchmark application to a GPU, and present results showing that significant performance improvements were achieved (over the use of a CPU alone), ranging from a factor of 2 to a factor of 7.5, depending on the problem size and level of memory optimisation performed.

This is a Technical Report from the HPCx Consortium

© UoE HPCx Ltd 2008

Neither UoE HPCx Ltd nor its members separately accept any responsibility for loss or damage from the use of information contained in any of their reports or in any communication about their tests or investigations.

1 Introduction

Graphics Processing Units (GPUs), which commonly accompany standard Central Processing Units (CPUs) in consumer PCs, are special purpose processors designed to efficiently perform the calculations necessary to generate visual output from program data. Video games have particularly high rendering demands, and this market has driven the development of GPUs which, in comparison to CPUs, offer extremely high performance for the monetary cost.

Naturally, interest has been generated as to whether the processing power which GPUs offer can be harnessed for more general purpose calculations (see e.g. [1]). In particular, there is potential to use GPUs to boost the performance of the types of simulations commonly done on traditional HPC systems such as HPCx. There are challenges to be overcome, however, to realise this potential.

The demands placed on GPUs from their native applications are, however, usually quite unique, and as such the GPU architecture is quite different from that of the CPU. Graphics processing is inherently extremely parallel so can be highly threaded and performed on the large numbers (typically hundreds) of processing cores found in the GPU chip. The GPU memory system is quite different to the standard CPU equivalent system. Furthermore, the GPU architecture reflects the fact that graphics processing typically does not require the same level of accuracy and precision as scientific simulation. Specialised software development is currently required to enable applications to efficiently utilise the GPU architecture.

This report first gives a discussion on scientific computing on GPUs. Then, we describe the porting of an HPC benchmark application to the NVIDIA TESLA GPU architecture, and give performance results comparing to use of a standard CPU.

2 Background

2.1 GPUs

The key difference between GPUs and CPUs is that while a modern CPU contains a few high-functionality cores, GPUs typically contain 100 or more basic cores. GPUs also boast a larger memory bus width than CPUs which results in faster memory access. The GPU clock frequency is typically lower than that of a CPU, but this gap has been closing over the last few years.

Applications such as rendering are highly parallel in nature, and can keep the cores busy, resulting in a significant performance improvement over use of a standard CPU. For applications less susceptible to such high levels of parallelisation, the extent to which the available performance can be harnessed will depend on the nature of the application and the investment put into software development.

2.1.1 Architecture

This section introduces the architectural design of GPUs. NVIDIA's products are focussed on here but offerings from other GPU manufacturers, such as ATI, are similar.

Figure 1 illustrates the layout of a GPU. It can be seen that there are many processing cores (processors) to perform computation, each grouped into multiprocessors. There are

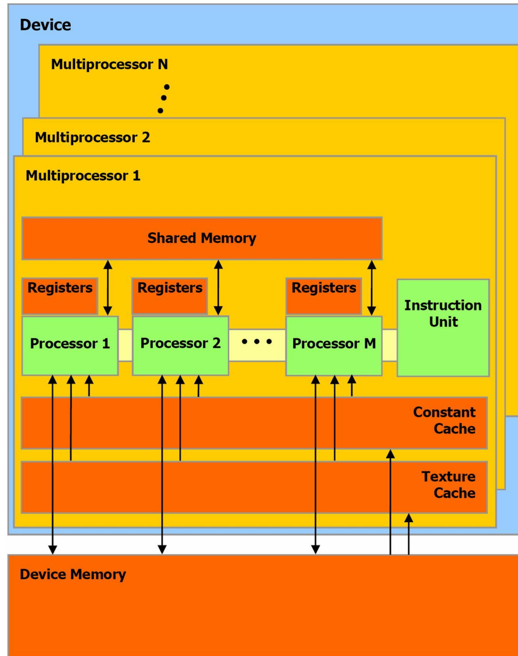

Figure 1: Architectural layout of NVIDIA GPU chip and memory. Reproduced with permission from [2].

several levels of memory which differ in terms of access speed and scope. The Registers have processor scope; the Shared Memory, Constant Cache and Texture Cache have multiprocessor scope and the Device (or Global) memory can be accessed by all cores on a chip. Note that the GPU memory address space is separate from that for the CPU, and copying of data between the devices must be managed in software. Typically, the CPU will run the program skeleton, and offload one or more computationally demanding code sections to the GPU. Thus, the GPU effectively *accelerates* the application. The CPU is referred to as the *Host* and the GPU as the *Device*. Functions that run on the Device are called *kernels*.

On the GPU, operations are performed by *threads* that are grouped into *blocks*, which are in turn arranged on a *grid*. Each block is executed by a single processor, however if there are enough resources available, several blocks can be active at the same time on a processor. The processor will time-slice the blocks to improve performance, one block performing calculations while another is waiting for a memory read, for example. Some of the memory available to the GPU exhibits considerably latency, however by using this method of time-slicing, this latency can be hidden for applications that are suitable.

A group of 32 threads is called a *warp*, and 16 threads a *half-warp*. GPUs achieve best performance when half-warps of threads perform the same operation. This is because in this situation, the threads can be executed in parallel. Conditionals can mean that threads do not perform the same operations and so they must be serialised. Such threads are said to be *divergent*. This also applies for Global Memory accesses: if the threads of a half-warp access Global Memory together and obey certain rules to qualify as being *coalesced*, then they access the memory in parallel and it will only take the time of a single access for all threads of the half-warp to access the memory.

Global Memory is located in the graphics card's GDDR3 memory. This can be accessed by all threads, although it is usually slower than on-chip memory. Memory access is significantly improved if memory accesses are coalesced (see 5.1.2.5 of the Programming Guide [2]) as this allows all the threads of a half-warp to access the memory simultaneously.

Shared Memory can only be accessed by threads in the same block. Because it is on-chip, the Shared Memory space is much faster than the local and Global Memory spaces. Approximately 16KB of shared memory are available on each MP (multi-processor), however to permit each MP to have several blocks active at a time (which improves performance) it is advisable to use as little Shared Memory as possible per block. A little bit less than 16KB is effectively available due to storage of internal variables. Shared Memory consists of 16 memory banks. When Shared Memory is allocated, each consecutive 32bit word is placed on a different memory bank. To achieve maximum memory performance, bank conflicts must be avoided (two threads trying to access the same bank at the same time). In the case of a bank conflict, the conflicting memory accesses are serialised, otherwise memory access by each half-warp is done in parallel.

Constant Memory is read-only memory that is cached. It is located in Global Memory, however there is a cache located on each Multi-processor. If the requested memory is in the cache, then access is as fast as Shared Memory, however if it is not then the access will be the same as a Global Memory access.

Texture Memory is read-only memory that is cached and is optimized for 2D spatial locality. This means that accessing [a][b] and [a+1][b], say, will probably get better speed than if [a][b] and [a+54][b] were accessed instead. The Texture Cache is 16KB per processor. This is a different 16KB to the Shared Memory, so using the Texture Cache does not reduce available Shared Memory.

Register memory exists and access speed is similar to Shared Memory. Each thread in a block has its own independent version of register variables declared. Variables that are too large will be placed in Local Memory which is located in Global Memory. The Local Memory space is not cached, so accesses to it are as expensive as normal accesses to Global Memory.

2.2 CUDA: Compute Unified Device Architecture

CUDA is a programming language developed by NVIDIA to facilitate writing programs that run on CUDA-enabled GPUs. It is an extension of C and is compiled using the *nvcc* compiler. The most commonly used extensions are *cudaMalloc** to allocate memory on the device, *cudaMemcpy** to copy data between the host and device and between different locations on the device,
kernel_name<<<grid dimensions, block dimensions >>>(parameters)
to launch a kernel, *threadIdx.x*, *blockIdx.x*, *blockDim.x*, and *gridDim.x* to identify the thread, block, block dimension, and grid dimension in the x direction.

CUDA addressed a number of issues that affected developing programs for GPUs, which previously required much specialist knowledge. CUDA is quite simple, so it will not take much time for a programmer already familiar with C to begin using it. CUDA also possesses a number of other benefits over previous methods of GPU programming. One of these is that it permits threads to access any location in the GPU memory and to read and write to as many memory locations as necessary. These were previously quite

limiting constraints, and so easing them represents a significant advantage for CUDA. Another major benefit is permitting access to Shared Memory, which was previously not possible.

To make adoption of CUDA as easy as possible, NVIDIA has created *CUDA U* [3] which contains a well-written tutorial with exercises as well as links to course notes and videos of CUDA courses taught at the University of Illinois. A Reference Manual and Programming Guide are also available [4].

The CUDA SDK [5] contains many example codes that can be used to test the installation of a GPU and, as the source codes are provided, demonstrate CUDA programming techniques. One of the provided codes is a template, providing the basic structure on which programs can be based.

One of the main features of CUDA is the provision of a Linear Algebra library (CUBLAS) and an FFT library (CUFFT). These greatly ease the implementation of many scientific codes on a GPU.

2.3 Review of GPU successes

In this section, some recent work involving using GPUs for scientific computing is highlighted.

- The Theoretical and Computational Biophysics group at the University of Illinois at Urbana-Champaign has used GPUs to achieve accelerations of between 20 and 100 times for molecular modelling applications. [6]

  The most time-consuming part of a molecular dynamics simulation is usually the evaluation of forces between atoms that do not share bonds. The UIUC group accelerated this portion of the calculation twenty times by performing it on a GPU rather than CPU. They hope to improve the performance of the entire calculation further by porting more of it to the GPU.

  Molecular dynamics simulations require the presence of appropriate ions. The placement of ions in certain circumstances can be very computationally demanding, requiring consideration of the electrostatic potential. Using a GPU to perform this placement resulted in a speed-up of one hundred times or more. An approximation method that the group implemented on the GPU resulted in an even further acceleration, while producing results very similar to those of the slower method.

  The group have also found that applications may scale better on GPUs than CPUs. A direct Coulomb summation program scaled to 4 GPUs with a scaling efficiency of 99.7%. As GPUs contain on-board high-performance memory, adding more GPUs does not reduce memory bandwidth for each core, unlike multi-core CPUs.

- Professor Mike Giles of Oxford University achieved a 100 times speed-up for a LIBOR Monte Carlo application and a 50 times speed-up for a 3D Laplace Solver. [7]

  The Laplace Solver was implemented on the GPU using only Global and Shared Memory. It uses a Jacobi iteration of a Laplace discretisation on a uniform 3D grid.

  The LIBOR Monte Carlo code used was quite similar to the original CPU code. It uses Global and Constant Memory.

- Many other UK researchers are also experimenting with GPUs. [8]
- NVIDIA has a showcase of applications reported to them. [9]
- GPGPU.org also maintains a list of researchers using GPUs. [10]
- RapidMind achieved a 2.4 times speed-up for BLAS SGEMM, 2.7 times for FFT, and 32.2 times for Black-Sholes. [11]

2.4 GPU Disadvantages and Alternative Acceleration Technologies

In this section, some disadvantages of the GPU architecture are discussed, and some alternative acceleration technologies are briefly described. The key limitation of GPUs is the requirement for a high level of parallelism to be inherent to the application to enable exploitation of the many cores. Furthermore, graphics processing typically does not require the same level of accuracy and precision as scientific simulation, and this is reflected in the fact that typically GPUs lack both error correction functionality and double precision computational functionality tends to be not available or incorporates a performance hit. This is expected to improve with future GPU architectures.

Another common criticism of GPUs is the large power consumption. The NVIDIA Tesla C870 uses up to 170W peak, and 120W typical. The amount of heat produced would make it difficult to cluster large numbers of GPUs together.

GPUs also place greater constraints on programmers than CPUs. To avoid significant performance degradation it is necessary to avoid conditionals inside kernels. Avoiding non-coalesced Global Memory accesses is very difficult for many applications, which can also severely degrade performance. The lack of any inter-block communication functionality means that it is not possible for threads in a block to determine when the threads in another block have completed their calculation. This means that if results of computation from other blocks are required then the only solution is for the kernel to exit and another launch, guaranteeing that all of the blocks have completed.

Finally, GPUs suffer from large latency in CPU-GPU communication. This bottleneck can mean that unless the amount of processing that is done on the GPU is great enough, it may be faster to simply perform calculations on the CPU.

There are other alternative acceleration technologies available, some of which are briefly described below.

Clearspeed One alternative to GPUs are processors designed especially for HPC applications, such as those offered by Clearspeed. These products are usually quite similar to GPUs, with a few modifications that usually make them more suitable for HPC applications. One of these differences is that all internal and external memory contains ECC (Error Correction Code) to detect and correct 'soft errors'. 'Soft errors' are random one-bit errors that are caused by external factors such as cosmic rays. In the graphics market such errors are tolerable, and so GPUs do not contain ECC, however for HPC applications it is often desirable or required. Clearspeed products also have more cores than GPUs, but they run at a slower clock speed to reduce heat loss. Double precision is also available.

Specialised products such as Clearspeed processors have a much smaller market than that of GPUs. This gives GPUs a number of advantages, such as economies of scale, greater availability, and more money spent on R&D.

Intel Larrabee Another alternative that is likely to generate much interest when it is released in 2009-2010 is Intel's Larrabee processor. This will be a many-core x86 processor with vector capability. It has the significant advantage over GPUs of making inter-processor communication possible. It should also solve a number of other problems that affect GPUs, such as the latency of CPU-GPU communication. It will initially be aimed at the graphics market, although specialised HPC products based on it are possible in the future. It is likely that it will also contain ECC to minimise 'soft errors'. AMD is also developing a similar product, currently named 'AMD Fusion', however few details have been released yet.

Cell Processor A Cell chip contains one Power Processor Element (PPE) and several Synergistic Processing Elements (SPEs). The PPE acts mainly to control the SPEs, which do most of the calculations. Cell processors are quite similar to GPUs. For some applications GPUs outperform Cell Processors, while for others the opposite is true.

FPGAs Field Programmable Gate Arrays (FPGAs) are programmable semiconductor devices that are based around a matrix of configurable logic blocks connected via programmable interconnects. As opposed to normal microprocessors, where the design of the device is fixed by the manufacturer, FPGAs can be programmed to compute the exact algorithm required by a given application. This makes them very powerful and versatile. The main disadvantages are that they are usually quite difficult to program, and they are also slow if high-precision is required. For certain tasks they are popular, however. Several time-consuming algorithms in Astronomy where only 4 bit precision is necessary are very suitable for FPGAs, for example.

3 GPU acceleration of an HPC benchmark

This section describes the porting of an HPC benchmark to the GPU architecture and gives a performance comparison against use of a CPU alone.

3.1 Benchmark description

The benchmark used in this performance analysis involves a reverse edge-detection: the Jacobi algorithm is used to reconstruct an image from its edge data. The computation involved is very similar in nature to many types of HPC applications, in particular those which solve a system of partial differential equations (for instance computational fluid dynamics applications). The benchmark, more simplistic than a full HPC application, is well understood at EPCC and has been used in a number of other performance evaluations (see e.g. [14] and [15]).

It is stressed that, although this benchmark relates to image processing, it is quite different in nature from the rendering computations for which GPUs are designed, and instead much more similar in nature to scientific applications.

This algorithm involves iterating the statement:

```
new[i][j]=(old[i+1][j]+old[i-1][j]+old[i][j+1]+old[i][j-1]-edge[i][j])/4
```

and after each iteration step copying *new* to *old*. The *edge* array holds the original edge data and *old* is initially set equal to *edge* with a border of zeros.

3.2 Architectures Used

A single NVIDIA TESLA C870 GPU [16] was used, which contains 128 cores running at 1.35Ghz. This was housed in a system containing a 1.8GHz AMD Opteron acting as the host processor.

The mechanism for GPU acceleration of an application involves running the code skeleton on the host and offloading dominant kernels to the GPU. To give a performance comparison, the benchmark should be run on the host CPU alone, and repeated utilising the GPU. However, since in this case the computational time on the host is negligible when utilising the GPU, we decided to instead compare to a more modern 2.6 GHz 64bit AMD Opteron processor in a separate system.

This TESLA GPU only supports single precision floating point arithmetic, so all benchmarking runs were done in single precision. Note also that we only utilised one core of the dual core Opteron, since we used a serial version of the benchmark. The use of OpenMP or Pthreads could have enabled use of the second core, thus improving the Opteron performance. However, for a real large scale parallel application, one could instead envisage using MPI in "virtual node mode", i.e. one MPI task per core, where each MPI task is accelerated by a separate GPU.

3.3 Implementation on GPU using CUDA

Different versions of the algorithm was implemented on the GPU (using CUDA), which differ in terms of the amount of effort invested in optimising for the GPU memory architecture.

3.3.1 v1: Using Global Memory

This algorithm can be implemented quite easily on a GPU with CUDA by using Global Memory. This can be achieved by allocating sufficient memory on the GPU using *cudaMalloc*, copying the required data to the device using *cudaMemcpy*, and then performing the calculations as normal (although the memory must be accessed using pointers). While access to global memory is very slow in comparison to other forms of memory available, the GPU time-slices blocks to hide this latency. Choosing block dimensions to be a multiple of 16 results in all memory reads and writes being coalesced, ensuring maximum global memory access speed is achieved. A more complicated addressing scheme is required for this implementation compared to others, which will decrease performance.

3.3.2 v2: Using Texture Memory

To make better use of the GPU's different types of memory, another version of the program was written. *old* and *edge* were created as CUDA Arrays. This is a proprietary storage format that can be used with the GPU's Texturing functionality, however it can

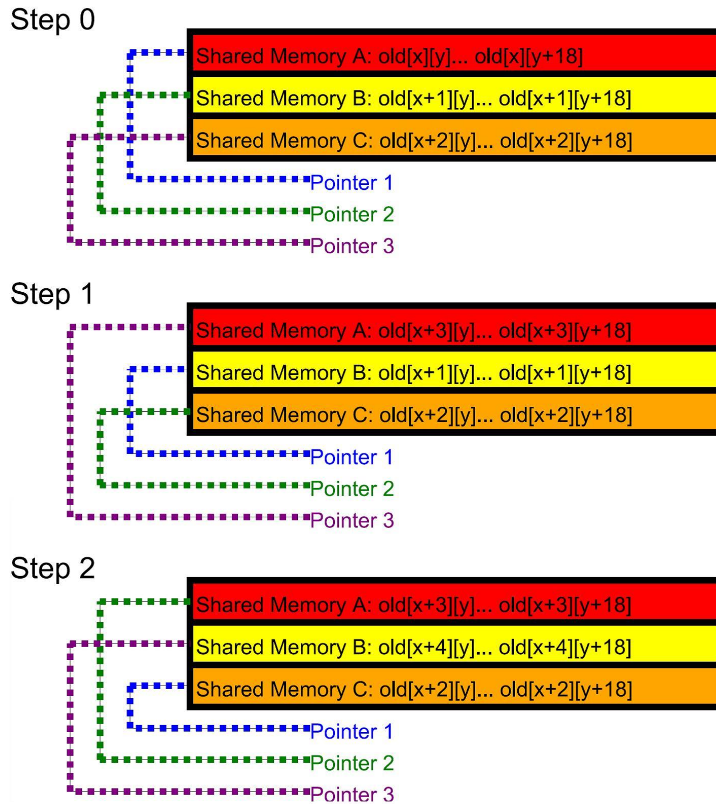

Figure 2: Swapping pointers to the columns in Shared Memory allows the block to progress across the image while loading only one new column from old at each step

only be written to through host runtime functions. The advantage of accessing memory through a texture reference is that there is little constraint on the size of the memory, as it is located in Global Memory, however accesses to it are likely to be faster than if it was just accessed as normal Global Memory as each Multiprocessor has a Texture Cache. Texture References are optimized for 2D spatial locality, so it should be particularly fast for this algorithm. While accesses to *new* also exhibit 2D spatial locality, it cannot be accessed through a Texture Reference as it must be written to from within the kernel, which is not possible for Textures. *new* was thus allocated using *cudaMallocPitch*, as is recommended by the Reference Manual for 2D arrays in Global Memory.

All reads from Global Memory using this method are through Texture References, which do not require coalescing, however it is necessary to ensure that the Global Memory write to *new* is coalesced. This can be achieved by reading columns of 16n + 2 elements from *old* (where n is an integer) which will allow 16n elements to be calculated. As the number of threads that are writing to *new* is a multiple of 16 and they obey all of the other conditions for coalesced access, it is possible for these writes to coalesced.

3.3.3 Further improvement

A further improvement was attempted that would reduce the amount of data that is loaded at the beginning of each iteration. To achieve this, 3 x (16n + 2) elements of *old* were loaded into Shared Memory and the calculation performed to determine the new values of the middle 16n elements, as before. In the previous version, the kernel would terminate at this point, but in the new version the pointer to column A of Shared Memory (see Fig 2) is modified to point to column B, the pointer to column B is modified to point to column C, and the pointer to column C is modified to point to column A. Following

this, the 16n elements of the column of *old* to the right of column C are loaded into column A of Shared Memory. There is now enough information stored in the Shared Memory to calculate the new values of the 16n elements of the column to the right of the previous column that was calculated. Repeating this process N times allows a single thread block to calculate the new values for a 16n x N area of the image during each iteration. The advantage of this method is that for each additional column that is calculated after the first, only one column of data needs to be loaded from *old*, in comparison to 3 columns being loaded for each column calculated in the previous version. Although this method probably reduces the amount of memory that must be read from Global Memory at each iteration, in experiments it did not run faster than the previous version. The reason for this is probably a combination of factors. The first of these is that this method reduces the number of blocks that are needed (as each block will cover a larger part of the image), which usually has a detrimental effect on performance as it reduces the ability of the processors to use their time-slicing capability. The values of n (which determines the number of threads per block) and N (which determines how far across the image each thread block should progress by swapping pointers during each iteration) must therefore be chosen carefully to ensure a balance between reducing memory loads and maintaining enough blocks to keep the processors active. The second reason is that storing pointers to the locations of the columns in Shared Memory increases the number of registers used per thread, which also reduces the number of blocks that can be simultaneously active. Two *__syncthreads()* calls must be made due to the Shared Memory usage, which has a detrimental effect on performance. Finally, the Texture Cache already does a good job of reducing Global Memory accesses, so this method will result in a much greater reduction. This version is not included in the following analysis.

3.3.4 Difficulties

The primary difficulty with this algorithm is that it requires information from neighbouring cells. This means that a 'border' of cells must be loaded around the cells that we wish to calculate. As no functionality exists in CUDA to allow communication between thread blocks, there is no way of passing the calculated values to blocks working on neighbouring cells, or of informing them when the updated value has been written to Global Memory. The only way the iteration can be performed is for the loop to be outside the kernel call. This ensures that all of the calculations from the previous step have been completed and the results written to memory, and so the next iteration is guaranteed to access the updated values, however it means that a lot of information must be discarded and then reloaded again, and that the entire image must be read from and then written back to Global Memory during each iteration step.

3.4 Results

Figure 3 shows the resulting benchmark times, for several array sizes, with and without GPU acceleration, where in the GPU case results are given for code versions 1 and 2 as described in section 3.3 [1]. For all but the lowest array sizes, the GPU performance significantly exceeds the CPU performance: by a factor of 2-3 for v1 and 5-7.5 for v2. The CPU performs better at the lowest array size relative to its performance for the

[1] All of the code run on the CPU and GPU was optimized with the compiler flag 'O3'

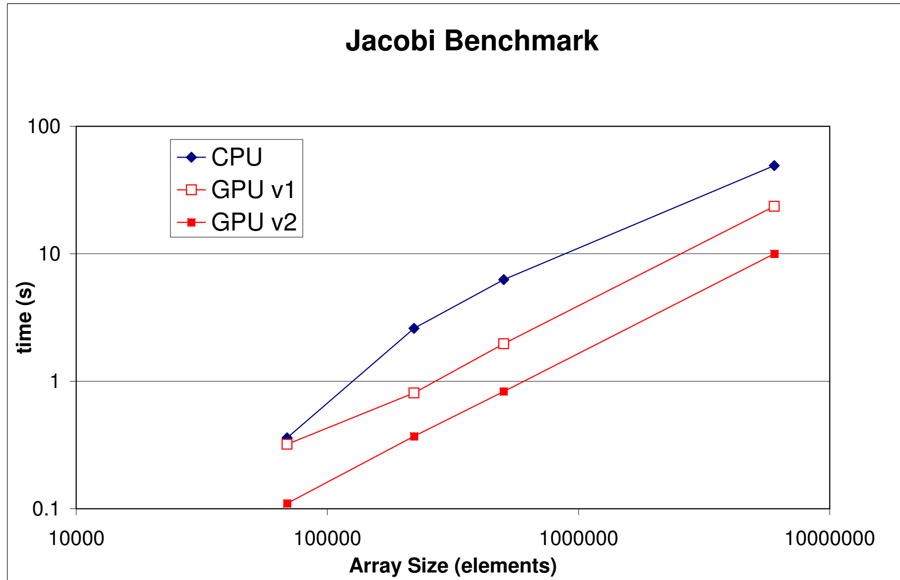

Figure 3: The time taken for 1000 iterations of the Jacobi algorithm, for various array sizes. Diamonds represent results from running on the CPU while open and closed squares represent code versions 1 and 2, respectively, run on the GPU.

larger array sizes, probably due to cache effects, but the GPU v2 still outperforms this by a factor of 3.2. It is expected that further optimisations, in particular algorithmic, would further boost the GPU performance.

4 Conclusions

GPUs, originally designed to satisfy the rendering computational demands of video games, potentially offer performance benefits for more general purpose applications, including HPC simulations. The differences between the GPU and standard CPU architectures result in the requirement that significant effort must be invested to enable efficient use of the GPU architecture for such applications.

We described the GPU architecture and methods used for software development, and reported that there is potential for the use of GPUs in HPC: there have been notable successes in several research areas. We described the porting of an HPC benchmark application to the GPU architecture, where several degrees of optimisation were performed, and benchmarked the resulting codes against code run on a standard CPU. The GPU was seen to offer up to a factor of 7.5 performance improvement.

5 Acknowledgements

We thank Paul Graham for help with initial investigations regarding general purpose programming on GPUs.

References

[1] General-Purpose Computation Using Graphics Hardware Web Site, http://www.gpgpu.org/

[2] NVIDIA CUDA Programming Guide, available from http://www.nvidia.com/object/cuda_develop.html

[3] CUDA U, http://www.nvidia.com/object/cuda_education.html

[4] NVIDIA CUDA Documentation, http://www.nvidia.com/object/cuda_develop.html

[5] CUDA SDK, http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html

[6] University of Illinois at Urbana-Champaign, GPU Acceleration of Molecular Modeling Applications, http://www.ks.uiuc.edu/Research/gpu/

[7] Mike Giles - High Performance Computing for Finance, http://people.maths.ox.ac.uk/~gilesm/hpc/

[8] OERC - UK academics using GPUs/accelerators, http://www.oerc.ox.ac.uk/research/many-core-and-reconfigurable-supercomputing/uk-academics-using-gpus-accelerators

[9] CUDA Zone, http://www.nvidia.com/cuda

[10] GPGPU Wiki People, http://www.gpgpu.org/w/index.php/GPGPU_People

[11] RapidMind, http://www.rapidmind.net/pdfs/RapidMindGPU.pdf

[12] NVIDIA CUDA Reference Manual, available from http://www.nvidia.com/object/cuda_develop.html

[13] Hewlett-Packard whitepaper on Accelerators For High Performance Computing, http://www.hp.com/techservers/hpccn/hpccollaboration/ADCatalyst/downloads/accelerators.pdf

[14] "Scientific Programming on the Cell using ALF", HPCx Technical Report 0708, 2007, Jon Hill http://www.hpcx.ac.uk/research/hpc/technicalreports/HPCxTR0708.pdf

[15] "Capability Computing, Achieving Scalability on over 1000 Processors", HPCx Technical Report 0301, 2003, Joachim Hein, Mark Bull http://www.hpcx.ac.uk/research/hpc/technicalreports/HPCxTR0301.pdf

[16] NVIDIA TESLA C870 Product Information http://www.nvidia.com/object/product_tesla_c870_us.html