

Experiences with Porting and Modelling Wavefront Algorithms on Many-Core Architectures

S.J. Pennycook, S.D. Hammond, G.R. Mudalige, and S.A. Jarvis

1 Introduction

We are currently investigating the viability of many-core architectures for the acceleration of wavefront applications and this report focuses on graphics processing units (GPUs) in particular. To this end, we have implemented NASA's LU benchmark [1] – a real world production-grade application – on GPUs employing NVIDIA's Compute Unified Device Architecture (CUDA) [2].

This GPU implementation of the benchmark has been used to investigate the performance of a selection of GPUs, ranging from workstation-grade commodity GPUs to the HPC "Tesla" and "Fermi" GPUs. We have also compared the performance of the GPU solution at scale to that of traditional high performance computing (HPC) clusters based on a range of multi-core CPUs from a number of major vendors, including Intel (Nehalem), AMD (Opteron) and IBM (PowerPC).

In previous work we have developed a predictive "plug-and-play" performance model of this class of application running on such clusters, in which CPUs communicate via the Message Passing Interface (MPI) [3, 4]. By extending this model to also capture the performance behaviour of GPUs, we are able to: (1) comment on the effects that architectural changes will have on the performance of single-GPU solutions, and (2) make projections regarding the performance of multi-GPU solutions at larger scale.

2 Wavefront Applications

Typical three-dimensional implementations of the parallel wavefront algorithm operate over a grid of $N_x \times N_y \times N_z$ grid-points. Computation starts at one of the grid's corner vertices and progresses to the opposite, which we refer to henceforth as a *sweep*.

By way of example, we consider a single sweep through the data-grid in which the computation required by each grid-point (i, j, k) is dependent upon the values of three neighbours: $(i-1, j, k)$, $(i, j-1, k)$ and $(i, j, k-1)$. In [5], Lamport showed that, for a given value of f , all grid-points that lie on the hyperplane defined by $i + j + k = f$ can be computed in parallel. Furthermore, all of the grid-points upon which this computation depends satisfy $i + j + k = f - 1$; by stepping in f and computing all satisfied grid-points, the dependency is preserved. We refer to each of these steps as a *wavefront step*.

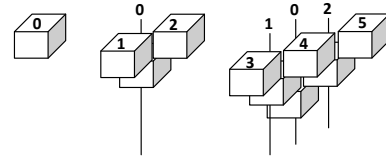


Figure 1: Two-dimensional mapping of threads onto a three-dimensional data grid.

This algorithm occupies large execution times at supercomputing centres such as the Los Alamos National Laboratory (LANL) in the United States and the Atomic Weapons Establishment (AWE) in the United Kingdom. Therefore, the acceleration of wavefront applications is of both commercial and academic interest.

2.1 GPU Implementation

Due to the strict data dependency present in wavefront applications, it is necessary to introduce a method of global thread synchronisation. We achieve this through the use of repeated kernel invocation, launching one kernel for the solution of each of the wavefront steps. In each kernel we launch $O(N^2)$ threads, mapping them to grid-points as shown in Figure 1. Those threads assigned to grid-points that are *not* currently on the active hyperplane exit immediately, without carrying out any computation.

Global memory is rearranged in keeping with this mapping (to ensure coalescence of memory accesses), but our kernels do not make use of shared memory.

2.2 GPU Performance

The graph in Figure 2 shows the performance of the GPU solution running on three HPC-capable GPUs: the Tesla T10, Tesla C1060 and Tesla C2050. Also included is the performance of the original Fortran benchmark executed on a quad-core 2.66GHz Intel "Nehalem" X5550 with 12GB of RAM, with and without simultaneous multithreading (SMT). The execution times reported are for problem classes A, B and C of the benchmark, which use data grids of size 64^3 , 102^3 and 162^3 respectively.

The GPU solution outperforms the original Fortran benchmark for all three problem classes, with the Tesla T10/C1060 and C2050 being up to 2.3x and 6.9x faster respectively.

One of the most eagerly anticipated additions to NVIDIA's Fermi architecture was the inclusion of ECC memory. ECC

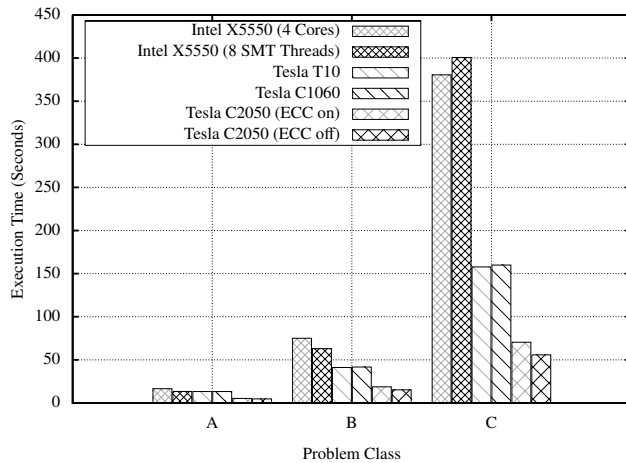


Figure 2: Execution times of LU across different workstation configurations.

memory is not without cost, however; firstly, enabling ECC on the Tesla C2050 decreases the amount of global memory available to the user from 3GB to 2.65GB; secondly it leads to a significant performance decrease. For a Class C problem run in double precision, execution times are almost 1.3x lower when ECC is disabled.

Though these results illustrate the performance benefits of GPU utilisation at the level of a single workstation, they do not answer the question of whether GPUs are ready to be adopted in place of CPUs at the cluster level. The development of performance models is likely to aid us in attempting to answer this question. Furthermore, results from several HPC clusters are to appear in an SC 10 paper later this year.

3 Performance Modelling

Previous work by the group developed a performance model for wavefront applications running at scale on CPUs communicating via MPI [3, 4]. We attempt to adapt this model to be used with GPU clusters, made possible by its reusable and generic nature.

We can capture changes to compute behaviour by replacing the original CPU “grind time” parameter with a GPU sub-model, whilst the MPI-associated overheads can be incorporated into the message latency parameter.

It is worth noting that, in our experience, the new costs that result from data transfer across the PCI Express (PCIe) bus are unlikely to contribute significantly to the overall execution time in a realistic GPU-enabled application. A well-written application will not transfer the entire data grid back to the

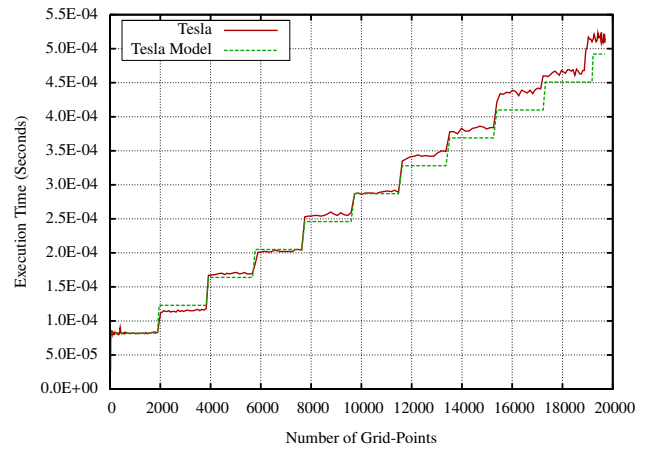


Figure 3: Execution times for Tesla C1060 and model predictions.

CPU for communication; this is wasteful, since only the border data is actually required. Packing and unpacking the MPI buffers on the GPU avoids the transfer of unnecessary data and greatly decreases the effect of these overheads.

Several other studies have demonstrated that it is possible to accurately model the performance of select application kernels based on source code analysis, or through low-level hardware simulation of GPUs [6, 7]. Our previous performance models have been produced at a higher level of abstraction, usually based on timing results from instrumented code and/or benchmark results. However, the concept of executing kernels on a separate device (shared by both CUDA and OpenCL) does not often lend itself well to such instrumentation. The best that we can achieve with CPU timers is to model the time taken for a given application kernel – in our case, this corresponds directly to the time taken for a given hyperplane step.

The graph in Figure 3 shows the execution times for each of the hyperplane steps in our GPU implementation of LU when run on a Tesla C1060 card. The first hyperplane step computes the value of a single grid-point, the second the values of three grid-points, the third six grid-points and so on, up until a maximum of approximately 20,000 grid points.

We model the execution time for a given number of grid-points, g , thus:

$$B(g) = \lceil g / (S \times T) \rceil \quad (1)$$

$$t(g) = h + (C \times (B(g) - 1) \times h) \quad (2)$$

where B is the number of blocks per stream multiprocessor (SM), S is the number of SMs and T is the number of threads per block. h represents the time taken to process the first active grid-point (calculated through code instrumentation). Finally, the presence of C attempts to compensate for the GPUs' ability to run several thread blocks concurrently. We currently know little more about this "concurrency factor" other than that it is a function of the number of thread blocks that can be scheduled to each SM, itself a function of the number of registers used by a particular kernel – we expect that the exact nature of this term will become clearer as our work progresses.

Essentially, the model states that the execution time of the kernel will stay constant so long as the number of thread blocks assigned to each SM remains the same. An increase in the number of blocks scheduled to each SM will increase the execution time by some factor (based on the ability to hide memory latency via time-slicing); any further blocks scheduled to an SM after they have been saturated will require additional processing steps and we model these as occurring serially. $S \times T$ threads are required to fill all SMs once, and the number of threads required to fully saturate all SMs is dependent upon register usage.

The wavefront kernel used in our experiments to date uses 107 registers, limiting the number of concurrent blocks per SM to 2. The Tesla C1060 has 30 SMs, and each thread block contains 64 threads. Our model predicts a small increase in execution time every $30 \times 64 = 1920$ threads and a larger increase every $30 \times 64 \times 2 = 3840$ threads, assumptions which both match up to the graph. However, our model is less accurate for large numbers of grid-points, which we believe to be the result of unforeseen memory contention issues not yet covered by our parameterisation (e.g. partition capping). For this kernel, we have found 0.5 to be an acceptable value of C .

Figure 4 shows the corresponding graph of execution times for a Tesla C2050 card, built on the "Fermi" architecture. It should be noted that the y -axis scale is not the same as the previous graph – the Fermi card is consistently around 2 - 3 \times faster than the Tesla. As before, the increases in execution time correspond to increases in the number of blocks scheduled to each SM. The GPU has 14 SMs, each supporting a maximum of 8 thread blocks, suggesting an increase in execution time every $14 \times 64 \times 8 = 7168$ threads. However, the remainder of the execution time graph appears to be linear in nature.

The different performance behaviour of Fermi is likely to be due to the architectural improvements made by NVIDIA, including a dual-warp scheduler and a two-tier hardware cache.

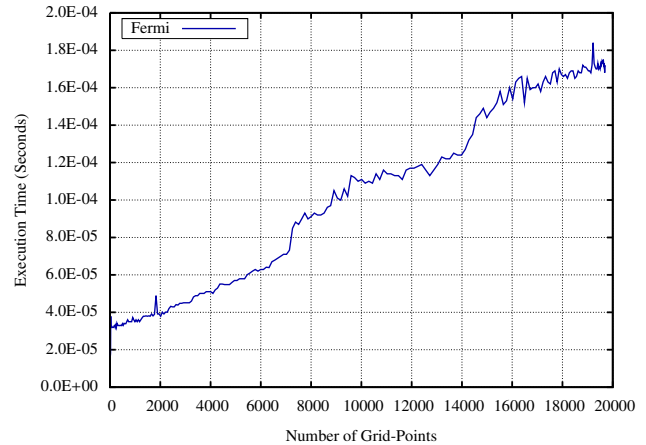


Figure 4: Execution times for Tesla C2050.

In future work, we intend to extend our existing Tesla model to GPUs based on the Fermi architecture.

References

- [1] D. Bailey, T. Harris, W. Saphir, R. Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. Technical Report NAS-95-020, NASA, December 1995.
- [2] The NVIDIA Compute Unified Device Architecture. <http://www.nvidia.com/cuda/>, 2010.
- [3] G. R. Mudalige, M. K. Vernon, and S. A. Jarvis. A Plug-and-Play Model for Evaluating Wavefront Computations on Parallel Architectures. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008)*. IEEE Computer Society, April, 2008.
- [4] G. R. Mudalige, S. A. Jarvis, D. P. Spooner, and G. R. Nudd. Predictive Performance Analysis of a Parallel Pipelined Synchronous Wavefront Application for Commodity Processor Cluster Systems. In *Proc. IEEE International Conference on Cluster Computing - Cluster2006*, Barcelona, September 2006. IEEE Computer Society.
- [5] L. Lamport. The Parallel Execution of DO Loops. *Commun. ACM*, 17(2):83–93, 1974.
- [6] S.S. Baghsorkhi, M. Delahaye, S.J. Patel, W.D. Gropp, and W.W. Hwu. An Adaptive Performance Modeling Tool for GPU Architectures. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Computing*, pages 105–114. ACM, 2010.
- [7] S. Hong and H. Kim. An Analytical Model for a GPU Architecture with Memory-Level and Thread-Level Parallelism Awareness. *ACM SIGARCH Computer Architecture News*, 37(3):152–163, 2009.