Performance Analysis of a Hybrid MPI/CUDA Implementation of the NAS-LU Benchmark

S.J. Pennycook, S.D. Hammond and S.A. Jarvis
Department of Computer Science
University of Warwick, UK
sjp@dcs.warwick.ac.uk

G.R. Mudalige
Oxford eResearch Centre
University of Oxford, UK
gihan.mudalige@oerc.ox.ac.uk

ABSTRACT

We present the performance analysis of a port of the LU benchmark from the NAS Parallel Benchmark (NPB) suite to NVIDIA's Compute Unified Device Architecture (CUDA), and report on the optimisation efforts employed to take advantage of this platform. Execution times are reported for several different GPUs, ranging from low-end consumer-grade products to high-end HPC-grade devices, including the Tesla C2050 built on NVIDIA's Fermi processor.

We also utilise recently developed performance models of LU to facilitate a comparison between future large-scale distributed clusters of GPU devices and existing clusters built on traditional CPU architectures, including a quad-socket, quad-core AMD Opteron cluster and an IBM BlueGene/P.

Keywords

LU, CFD, Wavefront, GPU, CUDA, Performance Model

1. INTRODUCTION

As the High Performance Computing (HPC) industry focuses on an exascale computing platform in the 2015-2018 timeframe, it is becoming clear that the design of such large systems is likely to lead to the development of radically different architectures to that of contemporary, general-purpose-processor machines. Apart from the challenge of physical size, the power consumption of an exascale system based on similar technologies to today would likely be significant – unless substantial improvements in efficiency are obtained over the coming years. Whilst power efficiencies in existing designs will undoubtedly appear, these alone are unlikely to be sufficient to enable an exascale reality; new architectures with new ways of constructing algorithms *will* be required.

One potential solution to these problems may come through the utilisation of "many-core" accelerators, such as general purpose graphics processing units (GPGPUs) or the upcoming "many-integrated core" (MIC) cards from Intel (*i.e.* Knight's Ferry and Knight's Corner). These devices feature large numbers of simple processing engines and/or the ability to simultaneously execute large numbers of threads in silicon real-estate which is comparable to existing processor designs. They therefore exhibit high levels of spatial and power efficiency for the amount of parallelism offered, providing more GFLOP/s of processing capability per Watt.

This high level of parallelism is not without cost however, as it results in a smaller amount of memory per-core (or per-thread) than would be found in traditional distributed memory solutions. Applications constructed under the assumption of few, high-power processing cores with large memories and high memory bandwidth are unlikely to port easily or exhibit good performance without some degree of modification. For organisations with limited low-level hardware expertise, or with few resources to port sizable legacy applications, selecting which optimisations and approaches to use in production is proving to be a difficult choice.

In this paper, we implement the LU benchmark from the NAS Parallel Benchmark (NPB) suite [5] on GPUs employing NVIDIA's Compute Unified Device Architecture (CUDA). The performance of the original unmodified FORTRAN 77 code is then compared to our implementation executing on single GPU devices and clusters of GPUs. The execution times presented in this paper are for the complete time to solution of the original problem – not a subset which has been identified as being particularly amenable to GPU acceleration. This paper makes the following contributions:

- We present the first known documented study of porting the LU benchmark to the CUDA GPU architecture. The benchmark constitutes a representative application regularly used in performance assurance and procurement studies for production-grade HPC environments around the world;
- The GPU-accelerated solution is executed on a selection of GPUs, ranging from workstation-grade, commodity GPUs to NVIDIA's HPC products. This is the first known work to feature an independent performance comparison of the Tesla C1060 and C2050 cards for a realistic scientific wavefront application;
- The performance of a single-GPU implementation is compared to that of the original FORTRAN 77 implementation from the NPB suite executing on a high-end Intel (Nehalem) processor, demonstrating the utility of GPU-acceleration at workstation scale;
- Performance modelling is used to compare GPU cluster performance to that of existing CPU clusters, including a quad-socket, quad-core AMD Opteron cluster and an IBM BlueGene/P;
- We present the first documented performance analysis of an LU Class E problem on petascale-capable GPU hardware. We aim to address the question of whether CPU- or GPU-based architectures are the best *current* technology for large wavefront problems.

The remainder of this paper is structured as follows: Section 2 discusses previous work; Section 3 describes the operation of the LU benchmark; Section 4 provides details of our GPU implementation; Section 5 presents a performance comparison of the CPU and GPU codes running on single workstations; Section 6 compares the execution time of our GPU implementation to that of traditional CPU clusters through the use of supporting performance models and simulations; finally, Section 7 concludes the paper.

2. RELATED WORK

The performance of pipelined wavefront applications is well understood for conventional multi-core processor based clusters [12, 17] and a number of studies have investigated the use of accelerator-based architectures for the Smith-Waterman string matching algorithm (a 2D wavefront application) [4, 18, 16]. However, performance studies for GPU-based implementations of 3D wavefront applications (either on a single device or at cluster scale) remain scarce; to our knowledge, this is the first such port of LU to a GPU.

Two previous studies [9, 19] detail the implementation of a different 3D wavefront application, the Sweep3D benchmark [1], on accelerator-based architectures. The first of these [19] describes an implementation of the Sweep3D benchmark that utilises the Cell B.E. The performance benefits of several optimisations are shown, demonstrating a clear path for the porting of similar codes to the Cell B.E. architecture.

In the second study [9], the Sweep3D benchmark is ported to a CUDA GPU and executed on a single Tesla T10 processor. Four stages of optimisation are presented: the introduction of GPU threads, using more threads with repeated computation, using shared memory and using a number of other methods that contribute only marginally to performance. The authors document good speed up, extrapolating that their GPU solution is almost as fast as the Cell B.E. implementation in [19].

These studies suggest that accelerator-based architectures are a viable platform for pipelined wavefront codes. However, one must be cautious when reading speedup figures: in some studies the execution times are presented for an optimised GPU code and an un-optimised CPU code [8]; in other work we do not see the overhead of transferring data across the PCI-Express (PCIe) bus [10]; in some the CPU implementation is serial [22], in others, parallel CPU and GPU solutions are run at different scales [13], or the CPU implementation is run on outdated hardware [9]. This is not the first paper to dispute such speedup figures [15, 23] and others have highlighted the importance of hand-tuning CPU code when aiming for a fair comparison [6, 7].

In this paper, performance comparisons are presented from two standpoints: (*i*) a single workstation comparison, where the performance of single GPUs is compared to that of single CPUs using all of the available cores, forming a full device-to-device comparison; and (*ii*) a strong-scaling study, comparing the performance of CPU and GPU clusters. We believe that this will allow comparison at both small and large scales and permits an exploration of the likely performance of future GPU clusters based on benchmarked performance data from existing cluster resources.

3. THE LU BENCHMARK

The LU benchmark belongs to the NAS Parallel Benchmark (NPB) suite, a set of parallel aerodynamic simulation benchmarks. The code implements a simplified compressible Navier-Stokes equation solver which employs a Gauss-Seidel relaxation scheme with symmetric successive over-relaxation (SSOR) for solving linear and discretised equations. The reader is referred to [5] for a discussion of the mathematics.

In practice, the three-dimensional data grid used by LU is of size N^3 (*i.e.* the problem is always a cube), although the underlying algorithm works equally well on grids of all sizes. As of release 3.3.1, NASA provide seven different application "classes" for which the benchmark is capable of performing verification: Class S (12^3), Class W (33^3), Class A (64^3), Class B (102^3), Class C (162^3), Class D (408^3) and Class E (1020^3). GPU performance results for Classes A through C are presented in Section 5; due to the significant resource demands associated with Classes D and E, benchmarked times and projections are shown for clusters in Section 6.

In the MPI implementation of the benchmark, this data grid is decomposed over a two-dimensional processor array of size $P_x \times P_y$, assigning each of the processors a stack of N_z data "tiles" of size $N_x/P_x \times N_y/P_y \times 1$. Initially, the algorithm selects a processor at a given vertex of the processor array which solves the first tile in its stack. Once complete, the edge data is communicated to two of its neighbouring processors. These adjacent processors – previously held in an idle state via the use of MPI-blocking primitives – then proceed to compute the first tile in their stacks, whilst the original processor solves its second tile. Once the neighbouring processors have completed their tiles, their edge data is sent downstream. This process continues until the last tile in the N_z dimension is solved at the opposite vertex to the original processor's starting tile, resulting in a "sweep" of computation through the data array.

Such sweeps, which are the defining features of pipelined wavefront applications, are also commonly employed in particle transport codes such as Sweep3D and Chimaera [17]. This class of algorithm is therefore of commercial as well as academic interest not only due to its ubiquity, but also the significant time associated with its execution at supercomputing sites such as NASA, the Los Alamos National Laboratory (LANL) in the US and the Atomic Weapons Establishment (AWE) in the UK.

The pseudocode in Algorithm 1 details the SSOR loop that accounts for the majority of LU's execution time.

Algorithm 1 Pseudocode for the SSOR Loop

```
for $iter = 1$ to $max\_iter$ do

    for $k = 1$ to $N_z$ do
        call jacld($k$)
        call blts($k$)
    end for

    for $k = N_z$ to 1 do
        call jacu($k$)
        call buts($k$)
    end for

    call l2norm()
    call rhs()
    call l2norm()

end for
```

Each of the subroutines in the loop exhibit different parallel behaviours: `jacld` and `jacu` are embarrassingly parallel and pre-compute the values of arrays then used in the forward and backward wavefront sweeps; `blts` and `buts` are re-

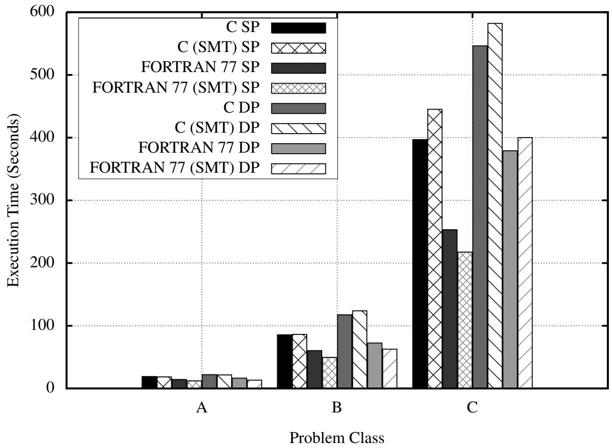


Figure 1: Execution Times for FORTRAN 77 and C

sponsible for the forward and backward sweeps respectively; `l2norm` computes a parallel reduction (on user-specified iterations); and `rhs` carries out three embarrassingly parallel stencil update operations. The number of loop iterations is configurable by the user at both compile- and run-time, but is typically 250 - 300 in Classes A through E.

4. GPU IMPLEMENTATION

Version 3.2 of the LU benchmark, on which our work is based, is written in FORTRAN 77 and utilises MPI for communication between processing-elements. The GPU implementation makes use of NVIDIA's CUDA. The standard language choice for developing CUDA programs is C/C++ and thus the first stage in our porting of LU was to convert the entire application to C. To provide a comparison of the performance trade-offs for CFD codes in using single or double precision floating-point arithmetic, the ported version of the benchmark was also instrumented to allow the selection of floating-point type at compile time.

Figure 1 shows a performance comparison of the original FORTRAN 77 code with our C port, running on an Intel X5550 in both single (SP) and double (DP) precision. The execution times given for the C port are prior to the use of a GPU (*i.e.* all mathematics are ported and running on only the CPU). As shown by the graph, the original FORTRAN 77 implementation is approximately 1.4x faster than our C port running on identical hardware. The fact that the C code is slower demonstrates that the performance improvements shown in Sections 5 and 6 come from the utilisation of the GPU, rather than from any optimisations introduced during the process of changing programming language.

At the time of writing, the maximum amount of memory available on a single CUDA GPU is 6GB (available in the Tesla C2070). At least 10GB is required to solve a Class D problem, thus the use of MPI is necessary if the code is to be able to solve larger and higher fidelity problems. Our CUDA implementation retains and extends the MPI-level parallelism present in the original benchmark, mapping each of the MPI tasks in the system to a single GPU.

The parallel computation pattern of wavefront applications involves a very strict dependency between grid-points. Lamport's original description of the Hyperplane algorithm in [14] demonstrated that the values of all grid-points on a given *hyperplane* defined by $i + j + k = f$ can be computed in parallel, with the parallel implementation looping over f rather than i, j and k. In order to ensure that the dependency is satisfied, it is necessary that we have a global synchronisation barrier between the solution of each hyperplane; since the CUDA programming model does not support global synchronisation *within* kernels, each hyperplane must be solved by a separate kernel launch.

4.1 Minimisation of PCIe Transfers

We port all subroutines within the loop to the GPU, thus eliminating the need to transfer memory between the device and host whilst iterating; the entire solution grid is transferred to the GPU prior to the SSOR loop, and the results are transferred back to the CPU when it is complete.

A similar optimisation is required for efficient utilisation of MPI. Since the CUDA card itself does not have access to the network, any data sent via MPI must first be transferred to the CPU and, similarly, any data received via MPI must be transferred to the GPU. Packing / unpacking the MPI buffers on the CPU requires a copy of the entire data grid – doing this on the GPU significantly decreases the data sent across the PCIe bus and allows elements of the MPI buffers to be packed / unpacked in parallel.

In the CPU implementation of LU, each call to `blts` or `buts` processes a single tile of grid-points of size $N_x/P_x \times N_y/P_y \times 1$, stepping through i and j in turn for some fixed k. Our GPU implementation can process grid-points in blocks of arbitrary depth (*i.e.* $N_x/P_x \times N_y/P_y \times kblock$), which has two advantages: (*i*) the number of MPI messages (and PCIe transfers) in each iteration decreases, at the expense of sending larger messages; and (*ii*) GPU parallel resources are utilised more efficiently, since 3D wavefronts feature larger hyperplanes than their 2D counterparts. However, setting *kblock* too high causes delay to downstream GPUs (which are waiting on data). Due to this performance trade-off, it is necessary to identify the optimal value of *kblock* through empirical evaluation or modelling.

4.2 Coalescence of Memory Accesses

We ensure that each memory access is *coalesced*, in keeping with the guidelines in [21]. The simplest way to achieve this is to ensure that all threads access contiguous memory.

However, the embarrassingly parallel sections of the code require a different memory arrangement to the wavefront sections. Therefore, we launch a rearrangement kernel between these sections to swap memory layouts. Though this kernel involves some uncoalesced memory accesses, which is largely unavoidable since it reads from one memory layout and writes to another, the penalty is incurred only once – rather than for every memory access within the methods themselves. The lack of global synchronisation within kernels prevents memory rearrangement in place, so we make use of a separate rearrangement buffer on the GPU.

4.3 Memory/Compute Tradeoffs

In the original benchmark, the `jacld` and `jacu` subroutines use the values of a grid-point and three neighbours (20 values total) to calculate 100 values per grid-point, which are then stored in four arrays. These values are then read by the `blts` and `buts` subroutines, where they are used in relatively few arithmetic operations. The high latency of GPU global memory makes these accesses particularly expensive.

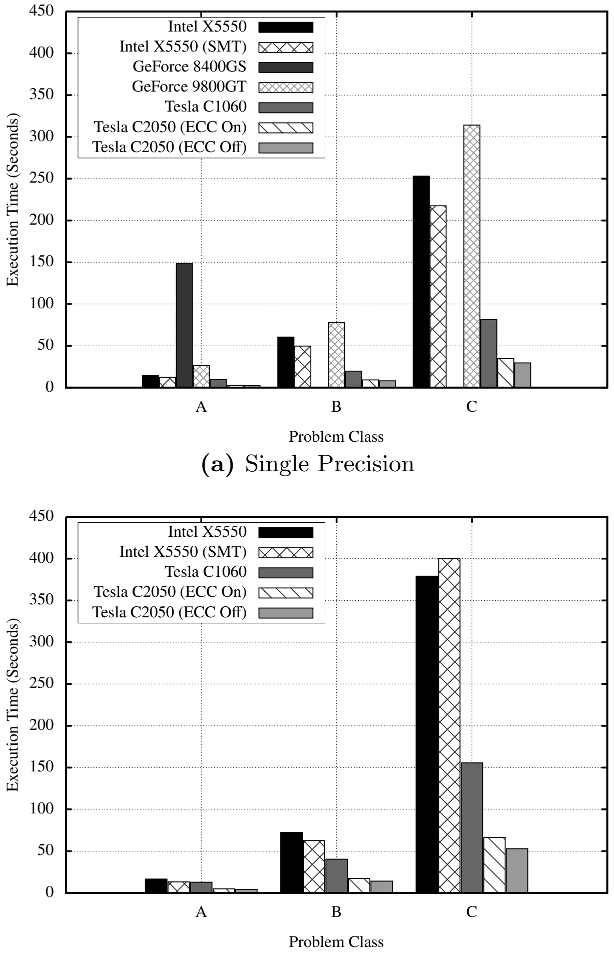
By reimplementing the `jacld` and `jacu` subroutines as four templated functions (with 25 template options each),

	GeForce		Tesla	
	8400GS	**9800GT**	**C1060**	**C2050**
Cores	8	112	240	448
Clock Rate	1.40GHz	1.38GHz	1.30GHz	1.15GHz
Global Memory	0.25GB	1GB	4GB	3GB[*]
Shared Memory	16kB	16kB	16kB	16kB[†]
Compute Capability	1.1	1.1	1.3	2.0

Table 1: Specification of NVIDIA GPUs Used

Device	Compiler	Options
Intel X5550 (Fortran)	Sun Studio 12 (Update 1)	`-O5 -native -xprefetch -xunroll=8 -xipo -xvector`
Intel X5550 (GPU Host)	GNU 4.3	`-O2 -msse3 -funroll-loops`
GeForce 8400GS/9800GT	NVCC	`-O2 -arch="sm_11"`
Tesla C1060/C2050	NVCC	`-O2 -arch="sm_13"`

Table 2: Configuration for Workstation Experiments



(a) Single Precision

(b) Double Precision

Figure 2: Workstation Execution Times

the `blts` and `buts` kernels need only retrieve the original 20 array values from memory, before computing the 100 values at the point in code where they are required. In addition to decreasing the number of memory accesses in our kernels, this decreases the amount of memory required (by removing the need to store the 100 calculated values between kernels).

5. WORKSTATION PERFORMANCE

Our first set of experiments investigates the performance of a single workstation executing the LU benchmark in both single and double precision. Classes A, B and C were executed on a traditional quad-core CPU, along with a range of consumer and high-end NVIDIA GPUs, including a Tesla C2050 built on the newest "Fermi" architecture. The full hardware specifications of the GPUs used in these experiments can be found in Table 1. The CPUs used in all of the workstations are "Nehalem"-class 2.66GHz Intel Xeon X5550s, with 12GB of RAM and each has the ability to utilise two-way simultaneous multi-threading (SMT). Table 2 lists the compiler configurations used for each platform.

The graphs in Figures 2a and 2b show the resulting execution times in single and double precision respectively. Due to their compute capability, the GeForce consumer GPUs used in our experiments appear in the single precision comparison only. It is clear from these graphs that the GPU solution outperforms the original FORTRAN 77 benchmark for all three problem classes when run on HPC hardware; the Tesla C1060 and C2050 are approximately 2.5x and 7x faster than the original benchmark run on an Intel X5550.

Our results also demonstrate the architectural improvements made by NVIDIA between each revision of the CUDA architecture: for our GPU implementation of the LU benchmark in single precision, the 9800GT is 8x faster than the 8400GS; the C1060 is 4x faster than the 9800GT; and the C2050 is 2.5x faster than the C1060. These performance gains are *not* simply the result of an increased core count; although the C2050 has only 4x as many cores as a 9800GT (and the clock rate of those cores is lower), it is approximately 10x faster. Improved memory bandwidth, relaxed criteria for coalescence of memory accesses and the introduction of an L2 cache are among the hardware changes we believe are responsible for these improvements.

Although the execution times of both consumer cards are worse than that of the Intel X5550, we believe that the performance of more recent consumer cards with higher compute capability (*e.g.* a GeForce GTX 280, 480 or 580) would be more competitive; it is our understanding that the difference between the consumer and HPC cards is largely one of memory and quality control, suggesting that the performance of newer consumer cards may be closer to the CPU.

Unexpectedly, the performance hit suffered by moving from single to double precision in the CPU and GPU implementations of the benchmark is comparable. This is surprising because, according to NVIDIA documentation, the ratios of double to single precision performance for the Tesla C1060 and C2050 are 12:1 and 2:1.

Our results also demonstrate that the C2050's ECC memory is not without cost; firstly, and as shown in Table 1, enabling ECC decreases the amount of global memory avail-

[*] 2.65GB with ECC-enabled.

[†] 48kB is available if the programmer selects higher shared memory instead of larger L1 cache.

			Actual			Predicted	
Machine	Nodes	Min.	Mean	Max.	Analytical	Sim. (No Noise)	Sim. (w/ Noise)
GPU Cluster (Class C)	1 (1 × C1060)	153.26	153.30	153.37	153.26	147.15	147.15
	4 (4 × C1060)	67.06	67.25	67.58	70.45	66.43	69.32
	8 (8 × C1060)	52.72	52.92	53.08	52.72	50.50	54.05
	16 (16 × C1060)	44.29	44.46	44.51	44.47	42.85	46.08
GPU Cluster (Class D)	4 (4 × C1060)	1359.93	1367.65	1372.85	1417.57	1375.28	1393.84
	8 (8 × C1060)	735.53	736.60	737.47	744.24	723.83	745.47
	16 (16 × C1060)	414.31	414.97	415.45	424.65	413.13	433.10
AMD Cluster (Class C)	1 (16 Cores)	220.58	224.00	228.98	176.63	217.62	227.38
	2 (32 Cores)	89.75	95.74	104.14	93.84	116.24	124.67
	4 (64 Cores)	66.41	73.26	83.41	50.37	62.54	69.44
	8 (128 Cores)	42.48	44.98	46.28	28.74	35.83	41.19
AMD Cluster (Class D)	64 (1024 Cores)	142.37	152.50	162.57	87.34	89.41	124.54
	128 (2048 Cores)	100.09	105.54	111.20	53.03	54.23	88.23
BlueGene/P (Class C)	32 (128 Cores)	59.44	59.45	59.45	57.76	55.63	58.32
	64 (256 Cores)	34.45	34.28	34.33	32.14	29.91	32.37
	128 (512 Cores)	20.89	20.92	20.94	19.50	17.45	19.53
	256 (1024 Cores)	13.84	13.86	13.87	12.12	10.33	11.98
BlueGene/P (Class D)	32 (128 Cores)	920.16	920.20	920.26	934.26	922.67	938.85
	64 (256 Cores)	474.67	474.75	474.80	486.79	476.34	489.94
	128 (512 Cores)	266.61	266.66	266.70	253.56	244.64	255.02
	256 (1024 Cores)	144.46	144.55	144.60	133.12	126.22	133.77

Table 3: Validations for the GPU Cluster, the AMD Cluster and the BlueGene/P (Times in Seconds)

able to the user from 3GB to 2.65GB; secondly, it leads to a significant reduction in performance. For a Class C problem run in double precision, execution times are almost 15% lower when ECC is disabled.

6. PERFORMANCE AT SCALE

The substantial rate at which the compute performance of GPUs has improved has lead to significant interest in their use as an accelerator at cluster scale. Whilst a number of large machines utilising GPUs have recently appeared in the TOP500 supercomputer list [3], they have yet to become commonplace. Analysis of the likely performance of codes executing on large scale GPU-based clusters is of interest to organisations who want to assess whether porting to such an architecture will be of benefit.

The experimental systems used in this study are an IBM BlueGene/P (DawnDev) and a commodity AMD Opteron cluster (Hera), both located at LLNL. DawnDev is an IBM BlueGene/P system and thus follows the tradition of IBM BlueGene architectures: a large number of lower performance processors (850MHz PowerPC 450d), a small amount of memory (4GB per node) and a proprietary BlueGene torus high-speed interconnect. Hera on the other hand utilises densely packed quad-socket, quad-core nodes featuring high performance AMD Opteron 2.3GHz processors and 32GB of memory per node. Hera uses the InfiniBand DDR high-speed interconnect and exemplifies a typical large capacity resource (127 TFLOP/s peak).

Compiler options used in the benchmarking are as follows: for the BlueGene/P, IBM XLF with `-O4 -qunroll -qipa -qhot`; for the AMD Opteron cluster, PGI 8.0.1 with `-O4 -Munroll -Minline`; the setup for the GPU clusters remains as reported in Table 2.

6.1 Analytical Model

In order to assess the performance of LU, we employ a recently published reusable model of pipelined wavefront computations from [17], which abstracts the parallel behaviour common to all wavefront implementations into a generic model. When combined with a limited number of benchmarked input parameters, the model is able to accurately predict execution time on a wide variety of architectures.

For an in-depth explanation of the model and its parameters, the reader is referred to [17]. The behaviour of a code's wavefront section is captured using the following parameters: a grind-time per grid-point (W_g), which is used to calculate the compute time on a given processor prior to communication (W), and a time-per-byte, used in conjunction with message sizes ($MessageSize_{NS}$ and $MessageSize_{EW}$) to calculate the communication times between processors. The value of W_g can be obtained via benchmarking or a low-level hardware model; the time-per-byte can be obtained via benchmarking (for a known message size) or a network sub-model. A $T_{nonwavefront}$ parameter represents all compute and communication time spent in non-wavefront sections of the code and can similarly be obtained through a collection of benchmarks or sub-models.

Our application of the reusable wavefront model makes use of both benchmarking and modelling, with W_g recorded from benchmark runs and the message timings of all machines taken from architecture-specific network models. The network models use results from the SKaMPI [20] benchmark, executed over a variety of message sizes and core/node counts in order to account for contention.

For the GPU cluster, the network model was altered to include the PCIe transfer times associated with writing data to and reading data from the GPU. The PCIe latencies and bandwidths of both cards were obtained using the "bandwidthTest" benchmark provided in the NVIDIA CUDA SDK; the MPI communication times employed are taken from benchmarks on a small InfiniBand cluster.

6.2 Simulation

In order to verify our findings, we also employ a performance model based on discrete event simulation. We use the WARPP simulator [11], which utilises coarse-grained compute models as well as high-fidelity network modelling to enable the accurate assessment of parallel application be-

	Nodes	Time (s)	Power (kW)	Peak (TFLOP/s)
Tesla C1060	1024	367.84	192.31	79.87
Tesla C2050	128	330.27	30.46	65.92
BlueGene/P	2048	262.51	32.77	6.95
AMD Opteron	256	315.47	97.28	32.69

Table 4: Comparison for a Fixed Execution Time

haviour at large scale. A key feature of WARPP is that it also permits the modelling of compute and network noise through the application of a random distribution to compute or networking events. In this study, we present two sets of runtime predictions from the simulator: a standard, noiseless simulation; and a simulation employing noise in data transmission times (see Table 3). In these noise-based simulations, the network events have a Gaussian distribution (with a standard deviation consistent with benchmarked data) applied to MPI communications. The simulator is therefore able to create a range in communication costs, which reflect the delays caused by other jobs and background networking events.

6.3 Model Validation and Projections

Validations of the analytical model and simulations are presented in Table 3. The AMD cluster was heavily loaded and this is evident in the runtime results. Model accuracy varies between the machines, but exceeds 90% for most runs on the BlueGene/P and the GPU clusters; model accuracy is approximately 80% on the AMD-cluster when accounting for additional network noise.

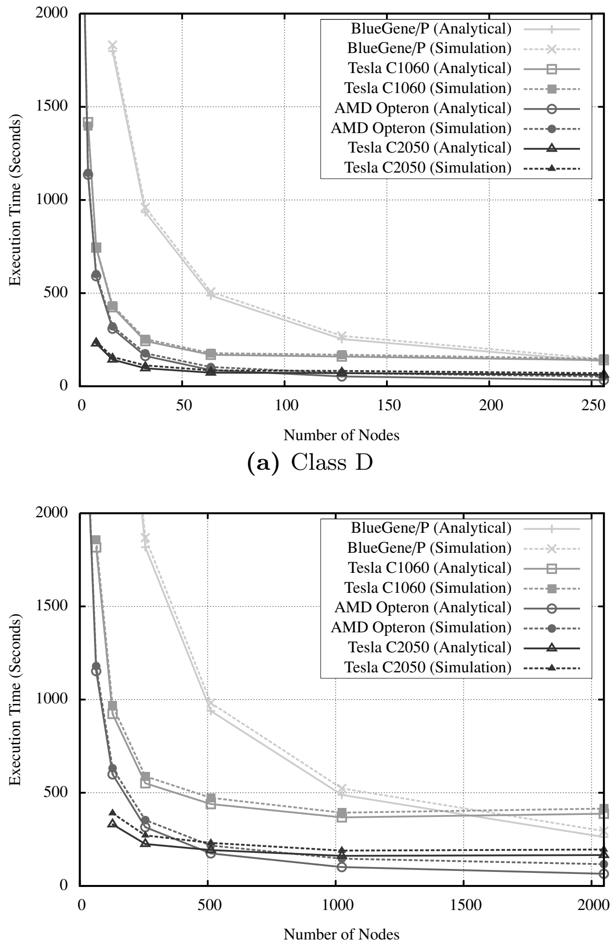
Such high levels of accuracy and correlation between the mathematical and simulation-based models – in spite of the presence of other jobs and background noise – provide a significant degree of confidence in predictive accuracy.

Figures 3a and 3b present model projections for the execution times of Class D and E problems, respectively. For each, projections are provided from the analytical model and from a simulation with network noise applied. Both modelling techniques provide similar runtime projections.

A cluster of Tesla C2050 GPUs provides the best performance at small scale for both problem sizes – up to 4x faster than a small AMD cluster and over 10x faster than a partially configured BlueGene/P rack. As the number of nodes increases, the BlueGene/P and AMD cluster demonstrate higher levels of scalability; the execution times for the CPU-based clusters continue to decrease as the number of nodes is increased, whilst those for the GPU clusters tend to plateau at a relatively low number of nodes. These scalability issues are not due to PCIe or MPI overheads, but rather the decreasing amount of parallelism per node associated with a decrease in subdomain size and *kblock* value.

Table 4 lists the power consumption (calculated from TDP) and theoretical peak for each of the four machines modelled. It should be noted that these power figures are for compute only and therefore do not include the power consumption of network switches or cooling – in the case of the GPU clusters, we also do not include the power consumption of the host CPU. Nonetheless, these figures are interesting and allow us to draw comparisons across the architectures.

Although the Tesla C2050 cluster consumes the least power, it also has the second highest theoretical peak processing rate. This demonstrates that although GPU-based solutions will no doubt continue to achieve high placements in rankings like the TOP500 [3] and Green500 [2], the level of sustained application performance may in fact be similar to that of existing cluster technologies – and lower as a percentage of theoretical peak.



(a) Class D

(b) Class E

Figure 3: Model Projections

7. CONCLUSIONS

Despite the impressive performance gains over a single quad-core Intel Nehalem and the considerable claims of the power of GPU devices elsewhere in academic literature, the results in this paper demonstrate that achieving performance at scale is still a challenge with such devices. For clusters of GPUs, the effects of data movement and decreasing amounts of parallelism per node must be considered if applications are to scale well.

This paper provides, with quantitative evidence, an insight into the performance that we might expect from future accelerator-based architectures; our results demonstrate that existing machines offer much lower peak performance than that expected of future GPU-based clusters, but equivalent sustained application performance. Accelerator-based machines will clearly provide exceptional advances in compute ability, but the performance gains demonstrated for single workstations will not easily be achieved at larger scales.

Our results also raise interesting questions about the future direction of HPC architectures. On the one hand, we might expect to see smaller clusters of SIMD or GPU nodes which will favour kernels of highly vectorisable code; on the other, we might expect highly parallel solutions typified by the BlueGene/P, where "many-core" will mean massively parallel quantities of independently operating cores. Therefore, the choice that application programmers will be faced with is one of focusing on low-level code design (to exploit instruction-level and thread-level parallelism) or higher-level, distributed scalability.

Acknowledgements

Access to the NVIDIA workstations is supported by the Royal Society Industry Fellowships. We are grateful to Scott Futral, Jan Nunes and the Livermore Computing Team for access to the DawnDev BlueGene/P and Hera machines located at the LLNL. The also thank the HPC team at Daresbury Laboratory (UK) for access to their GPU-cluster.

8. REFERENCES

[1] The ASCI Sweep3D Benchmark. `http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/asci_sweep3d.html`, 1995.

[2] The Green 500 List : Environmentally Responsible Supercomputing. `http://www.green500.org`, November 2010.

[3] Top 500 Supercomputer Sites. `http://www.top500.org`, November 2010.

[4] A. M. Aji and W. C. Feng. Accelerating Data-Serial Applications on GPGPUs: A Systems Approach. Technical Report TR-08-24, Computer Science, Virginia Tech., 2008.

[5] D. Bailey et al. The NAS Parallel Benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[6] R. Bordawekar, U. Bondhugula, and R. Rao. Believe it or Not! Multi-core CPUs Can Match GPU Performance for FLOP-intensive Application! Technical Report RC24982, IBM Research, April 2010.

[7] R. Bordawekar, U. Bondhugula, and R. Rao. Can CPUs Match GPUs on Performance with Productivity?: Experiences with Optimizing a FLOP-intensive Application on CPUs and GPU. Technical Report RC25033, IBM Research, August 2010.

[8] M. Boyer, D. Tarjan, S. T. Acton, and K. Skadron. Accelerating Leukocyte Tracking using CUDA: A Case Study in Leveraging Manycore Coprocessors. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, May 2009.

[9] C. Gong, J. Liu, Z. Gong, J. Qin, and J. Xie. Optimizing Sweep3D for Graphic Processor Unit. In *Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing*, May 2010.

[10] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High Performance Discrete Fourier Transforms on Graphics Processors. In *Proceedings of the ACM/IEEE Supercomputing Conference*, November 2008.

[11] S. D. Hammond, G. R. Mudalige, J. A. Smith, S. A. Jarvis, J. A. Herdman, and A. Vadgama. WARPP: A Toolkit for Simulating High-Performance Parallel Scientific Codes. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, March 2009.

[12] A. Hoisie, O. Lubeck, H. Wasserman, F. Petrini, and H. Alme. A General Predictive Performance Model for Wavefront Algorithms on Clusters of SMPs. In *Proceedings of the International Conference on Parallel Processing*, August 2000.

[13] D. A. Jacobsen, J. C. Thibault, and I. Senocak. An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters. In *Proceedings of the 48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, January 2010.

[14] L. Lamport. The Parallel Execution of DO Loops. *Communications of the ACM*, 17:83–93, February 1974.

[15] V. W. Lee et al. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, June 2010.

[16] S. Manavski and G. Valle. CUDA Compatible GPU Cards as Efficient Hardware Accelerators for Smith-Waterman Sequence Alignment. *BMC Bioinformatics*, 9(Suppl 2):S10, 2008.

[17] G. R. Mudalige, M. K. Vernon, and S. A. Jarvis. A Plug-and-Play Model for Evaluating Wavefront Computations on Parallel Architectures. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, April 2008.

[18] Y. Munekawa, F. Ino, and K. Hagihara. Design and Implementation of the Smith-Waterman Algorithm of the CUDA-Compatible GPU. In *Proceedings of the IEEE International Conference on Bioinformatics and Bioengineering*, October 2008.

[19] F. Petrini, G. Fossum, J. Fernández, A. L. Varbanescu, M. Kistler, and M. Perrone. Multicore Surprises: Lessons Learned from Optimizing Sweep3D on the Cell Broadband Engine. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, July 2007.

[20] R. Reussner, P. Sanders, L. Prechelt, and M. Müller. SKaMPI: A Detailed, Accurate MPI Benchmark. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 492–492, 1998.

[21] S. Ryoo et al. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, February 2008.

[22] T. Shimokawabe et al. An 80-Fold Speedup, 15.0 TFlops Full GPU Acceleration of Non-Hydrostatic Weather Model ASUCA Production Code. In *Proceedings of the ACM/IEEE Supercomputing Conference*, November 2010.

[23] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. E. Guney, and A. Shringarpure. On the Limits of GPU Acceleration. In *Proceedings of the USENIX Workshop on Hot Topics in Parallelism*, June 2010.