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

S.J. Pennycook, S.D. Hammond and S.A. Jarvis
Performance Computing and Visualisation
Department of Computer Science
University of Warwick, UK
{sjp, sdh, saj}@dcs.warwick.ac.uk

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

ABSTRACT

The emergence of Graphics Processing Units (GPUs) as a potential alternative to conventional general-purpose processors has led to significant interest in these architectures by both the academic community and the High Performance Computing (HPC) industry. While GPUs look likely to deliver unparalleled levels of performance, the publication of studies claiming performance improvements in excess of 30,000x are misleading. Significant on-node performance improvements have been demonstrated for code kernels and algorithms amenable to GPU acceleration; studies demonstrating comparable results for full scientific applications requiring multiple-GPU architectures are rare.

In this paper we present an analysis of a port of the NAS LU benchmark to NVIDIA's Compute Unified Device Architecture (CUDA) - the most stable GPU programming model currently available. Our solution is also extended to multiple nodes and multiple GPU devices.

Runtime performance on several GPUs is presented, ranging from low-end, consumer-grade cards such as the 8400GS to NVIDIA's flagship Fermi HPC processor found in the recently released C2050. We compare the runtimes of these devices to several processors including those from Intel, AMD and IBM.

In addition to this we utilise a recently developed performance model of LU. With this we predict the runtime performance of LU on large-scale distributed GPU clusters, which are predicted to become commonplace in future high-end HPC architectural solutions.

Categories and Subject Descriptors

B.8.2 [**Performance Analysis and Design Aids**]; C.1.2 [**Multiple Data Stream Architectures**]

General Terms

GPU, Performance

Keywords

LU, CFD, Wavefront, GPU, Performance Model

1. INTRODUCTION

As the High Performance Computing (HPC) industry begins to focus on the aspiration of an exa-scale system 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 that would be required for an exa-scale system based on similar technologies to today would likely be significant unless substanial improvements in efficiency were to be obtained over the coming years. Whilst power efficiencies in existing designs will undoubtedly appear, these alone are unlikely to be sufficient to enable an exa-scale reality - new architectures with new ways of crafting algorithms *will* be required.

One potential solution to these exa-scale problems may come through the utilisation of many-core architectures. Typified by devices such as general-purpose graphics processing units (GP-GPUs), streaming processors such as IBM's Cell Broadband Engine (Cell B.E.) or the forthcoming Knight's Ferry product from Intel, these devices feature large numbers of simple processing engines, 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, potentially providing more GFLOP/s of processing capability per Watt of energy consumed.

The higher levels of parallelism available in many-core architectures are, however, not without cost. Perhaps most importantly, they result in smaller amounts of memory per-core or per-thread (sometimes as low as megabytes or even kilobytes) – a stark contrast to MPI-based distributed memory solutions, where the memory per-core is typically several gigabytes and more easily extensible. As the core counts of these devices increase, application designers will be faced with lower memory bandwidth per-core making the transfer of data between processing engines relatively more expensive. Applications that have been constructed under the assumption of fewer, high-power processing cores with large memories and high memory bandwidth are unlikely to be easy to port or exhibit good performance without some degree of change.

Application designers who seek to port their codes to these novel architectures are also given false hope by some academic literature which provides examples of how best to modify their problems; the large speedups quoted in GPU studies, some as high as 30,000x [1], are simply unachievable for a large number of *real* scientific workloads. In many cases the techniques which are presented, although providing impressive speedups for the specific instance being studied, lack portability or are compared in an unrealistic or significantly biased context. For organisations with limited low-level GPU expertise, or with few resources to port sizable legacy applications, selecting which optimisations and approaches to use in production is becoming a difficult choice.

On the other hand, several studies have demonstrated that GPUs are viable devices for the acceleration of select application kernels, from domains such as Fourier analysis [6, 18], linear algebra [15, 24], parallel sorting [7, 9], computational biology [17, 21] and graph analysis [8, 12]. Considerable interest therefore remains in demonstrating *realistic* performance for these kernels when porting full applications and industry-grade benchmarks.

In this paper, we implement and investigate the performance of NASA's LU benchmark [2] – a representative HPC benchmark – on GPUs employing NVIDIA's Compute Unified Device Architecture (CUDA). We compare the performance of the unmodified original MPI-based implementation of the benchmark from the NAS Parallel Benchmark (NPB) suite to that of a version implemented and run on both single GPU devices and clusters of GPU devices. The execution times presented in this paper are for the complete time to solution of the original problem, rather than a subset particularly amenable to a GPU solution (as is the case in a number of studies). Specifically, this paper makes the following contributions:

- We present the first documented study of porting the LU benchmark to a GPU architecture. The application kernels within LU exhibit several different parallel behaviours, including a parallel wavefront dependency. The benchmark constitutes a representative application which is regularly used in performance assurance and procurement studies for production-grade HPC environments around the world. The GPU implementation is developed using NVIDIA's CUDA programming model.
- The GPU-accelerated solution is executed on a selection of GPUs, ranging from workstation-grade, commodity GPUs to those specifically designed by NVIDIA for HPC. This includes a Tesla C2050, built on the recently released NVIDIA "Fermi" architecture. This is the first known work to feature an independent performance comparison of these GPU device architectures for a realistic scientific wavefront application.
- We extend the single GPU solution to one that can be executed on a distributed cluster of GPUs, employing the Message Passing Interface (MPI) to communicate between separate CUDA cards. This solution permits scaling of the problem size to larger classes than would be possible on a single device and demonstrates the performance which users might expect from future HPC architectures where accelerators are deployed.
- The performance of the MPI/CUDA hybrid solution is compared with the original Fortran benchmark, with results collected from traditional HPC clusters based on a range of current market leading multi-core CPUs. Our comparison includes HPC processor offerings from a number of major processor vendors, including Intel (Nehalem), AMD (Opteron) and IBM (PowerPC). In comparing the results from these clusters we demonstrate the relative performance and scaling of these solutions as more nodes are added to the computation. These results go some way to independently addressing the question of whether CPUs or GPUs are the best current technology for solving large wavefront problems.
- Finally, we employ a recently developed reusable analytical model [16] of wavefront computations to predict the performance of GPU clusters at larger scales and larger problem sizes. We present the first assessment in academic literature of the likely performance of NAS LU's Class E problem on peta-scale GPU hardware.

The remainder of this paper is structured as follows: Section 2 discusses previous work related to this study; Section 3 describes the operation of the LU benchmark; Section 4 provides implementation details for our GPU port of LU; Section 5 presents a performance comparison of the CPU and GPU codes running on single workstations, housing NVIDIA GPUs of different compute capability; Section 6 compares the execution time of our GPU implementation to the original Fortran benchmark running on traditional CPU architectures and comments on the scalability of both solutions through the use of a supporting performance model; finally, Section 7 concludes the paper and provides suggestions for future work.

2. RELATED WORK

In this paper we present what is, to our knowledge, the first documented port of the LU benchmark to a GPU-based architecture. LU is part of a class of applications known as "pipelined wavefront computations", which all share a similar parallel computation and communication pattern. The performance of such applications is well understood for conventional multi-core processor based clusters [10, 16], but performance studies for GPU-based implementations (either on a single device or multiple devices) remain scarce.

Two previous studies [5, 20] detail the implementation of a different wavefront computation, the Sweep3D benchmark [22], on data-parallel architectures.

The first of these studies [20] accelerates Sweep3D through the utilisation of the Cell Broadband Engine (Cell B.E.) developed by IBM. The Cell B.E. port exploits five levels of parallelism: MPI level (present in the original CPU benchmark) between nodes, thread-level, data-streaming into the processor SPEs, vectorisation and, finally, pipelining. The performance benefits of implementing each of these parallelisation levels are shown in order, demonstrating a step-by-step path for porting similar codes to the Cell B.E. The resulting Cell B.E. implementation is shown to be approximately 4.5x and 5.5x times faster than an equivalent implementation running on a 1.9GHz POWER-5 processor and a 2.6GHz AMD Opteron processor respectively. It is not made clear how many cores the Opteron has, or whether the Cell B.E.'s performance is compared against a single processor core or multiple cores communicating via MPI.

In the second study [5], Sweep3D is ported to the GPU and executed on a single Tesla T10 processor from a Tesla S1070 server. The authors present four stages of optimisation: 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 speedup. The final speedup of the GPU implementation, compared against a serial solution running on a single core of a 2.40GHz Intel Core 2 Quad Q6660, is quoted as 2.25x. The authors conclude that this speedup is good, extrapolating that their GPU solution is almost as fast as the Cell B.E. implementation described in [20].

These studies suggest that data-parallel architectures are a viable alternative to traditional CPU architectures for the acceleration of scientific wavefront codes. However, it is difficult to use or compare the results directly due to the use of *speedup* as a method of evaluating parallel performance. This criticism is applicable to other GPU papers also; indeed, one of the speedups listed on the NVIDIA website [1] is 30,000x, and figures like these have recently been disputed by Intel [14]. IBM have also shown that optimised CPU code can match GPU performance [4]. The comparison between execution times is often made between an optimised GPU code and an unoptimised CPU code, or does not measure the overhead of transferring data across the PCI-Express (PCIe) bus. Worse still, the base CPU implementation is sometimes serial; in other cases, parallel CPU and GPU solutions are run at vastly different scales, or run on outdated hardware. Such comparisons are skewed in the GPU's favour and promote an unrealistic expectation of the performance benefits of these new architectures.

In [11], for example, a hybrid MPI/CUDA solution is presented for incompressible flow computations, with an investigation into the efficiency and scalability of the code executed on a multi-GPU cluster of 64 NVIDIA Tesla-based nodes. The authors report a 130x speedup from using 128 GPUs (a total of 30,720 CUDA cores) over an 8-core CPU solution using UNIX-pthreads, concluding that CFD simulations can be substantially accelerated through the utilisation of GPUs. In this paper, we propose an alternative view on the presentation of execution times (and speedups) which we believe to be more useful for the application programmer.

Specifically, we present two main studies: (1) a single workstation comparison where single GPUs are compared to single CPUs using all of their available cores, forming a device-to-device comparison and (2) a multiple node comparison in which the execution times of multiple GPU devices are compared to those of varying numbers of nodes from several leading HPC architectures. Besides being a much fairer comparison than those discussed in this section, this allows application developers to gain a better understanding of the likely performance of their code at small and large scale. It also permits an exploration of the likely performance of GPU-based ports based on the execution times of existing cluster resources.

Our optimised MPI/CUDA hybrid solution is compared to the original, optimised Fortran code as supplied by NASA, making the CPU results generalisable to those published elsewhere. Every effort has also been made to ensure that suitable compilers and compiler flags have been chosen and are reported in order to allow these results to be reproduced (see Table 2).

3. BACKGROUND

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 [3, 23] for a thorough discussion of the mathematics. Simplified, the code solves the $(n+1)^{th}$ time step of the discretised linear system:

$$U^{n+1} = U^n + \Delta U^n$$

using:

$$K^n \Delta U^n = R^n$$

where K is a sparse matrix of size $N_x \times N_y \times N_z$ and each of its matrix elements is a 5^2 sub-matrix. An SSOR-scheme is used to expediate convergence with the use of an over-relaxation factor $\delta \in (0, 2)$, such that:

$$U^{(n+1)} = U^n + (1/(\delta(2-\delta))\Delta U^n$$

The SSOR operation is re-arranged to enable the calculation to proceed via the solution of a regular sparse, block-lower (L) and upper (U) triangular system (giving rise to the name LU). The algorithm proceeds through the computing of the right-hand side vector R^n followed by the computing of the lower-triangular and then upper-triangular solutions. Finally the solution is updated.

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 D are presented in Section 5; due to the lengthy execution times and significant resource demands associated with Class E, projections of execution times are shown in Section 6.

In the MPI implementation of the benchmark, this data grid is decomposed over a two-dimensional processor array of size $n \times m$, assigning each of the processors a stack of N_z data "tiles" of size $N_x/n \times N_y/m \times 1$. 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 (which has been updated during this solve step) 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. Once the neighbouring processors have completed their tiles, their edge data too 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 the wavefront design pattern, are also commonly employed in particle transport benchmarks such as Sweep3D [22] and Chimaera [16]. 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 large supercomputing sites such as NASA, the Los Alamos National Laboratory (LANL) and the United Kingdom's Atomic Weapons Establishment (AWE). LU is simpler in operation than the particle transport codes in that it only executes two sweeps through the data array – one from the vertex at processor 0, and another in the opposite direction – as opposed to the eight sweeps found in both Sweep3D and Chimaera. Nonetheless, it provides an opportunity to study a representative science benchmark which, when scaled, can consume vast amounts of processing time (hundreds of thousands to millions of iterations) in its own right.

4. IMPLEMENTATION OF NAS-LU ON THE GPU

Version 3.2 of the LU benchmark, on which our work is based, is written in Fortran and utilises MPI for communication between processors. The GPU implementation makes use of NVIDIA's CUDA, since at the time of writing it is the most mature and stable programming model available for the development of GPU computing applications. The standard language choice for developing CUDA programs is C and, although the Portland Group offer a commerical alternative allowing the use of CUDA statements within Fortran applications, 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 instrumented to allow the selection of floating-point type at compile time. Although NASA explicitly requests double precision in the benchmark suite (with a maximum allowable error of 10^{-8}), we have included single-precision calculations to measure the performance of consumer GPU devices. The accuracy of these single-precision solutions is lower but the mathematics is otherwise identical and represents a coarse-grade calculation which might provide useful approximate solutions on low-end, lower-cost hardware.

Listing 1: The SSOR loop of LU.

```
for (iter = 1 to max_iter){
  for (k = 1 to nz){
    jacld(k)
    blts(k)
  }
  for (k = nz to 1){
    jacu(k)
    buts(k)
  }
  l2norm()
  rhs()
  l2norm()
}
```

At the time of writing, the maximum amount of memory available on a single CUDA GPU is 6GB (available in the Tesla C2070), and therefore the use of MPI is necessary if the code is to scale and be able to solve larger and higher fidelity problem sizes. Thus, our C implementation retains and supplements the MPI level of parallelism present in the original benchmark. Each of the MPI tasks in the system is mapped to a single CPU core, which is in turn responsible for controlling a single GPU. The pseudocode in Listing 1 details the SSOR loop that accounts for the majority of LU's execution time.

Each of the subroutines in the loop exhibit different parallel behaviours: `blts` and `buts` are responsible for the forward and backward sweeps respectively; `jacld` and `jacu` are embarassingly parallel, and pre-compute several values then used in the forward and backward sweeps; `l2norm` computes a parallel reduction; and `rhs` carries out three embarassingly parallel update operations. As this loop is typically repeated for around 250 - 300 iterations, all subroutines within the loop have been ported to the GPU to eliminate the need to transfer memory to and from the host.

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

The memory access patterns of the embarrassingly parallel routines (*i.e.* `rhs`, `l2norm`) are different to those of the sweep routines (*i.e.* `blts` and `buts`). In order to ensure best performance and that global memory accesses are *coalesced*, it is necessary to rearrange the memory layout of the solution arrays between calls to these routines. Due to the lack of global synchronisation within CUDA kernels, it is not possible to rearrange the memory in-place and so three copies of each solution array are stored on the GPU at any one time. For larger problem sizes running on a single GPU device, this requires a significant amount of memory. It may be possible to avoid this overhead through the use of pinned memory and more efficient data rearrangement kernels – we leave this to future work.

The current implementations of the embarrassingly parallel subroutines have been optimised for parallel performance. Each is launched on a grid of $N \times N \times N$ threads, where each thread is responsible for the computation required by a single grid-point. The execution time of the SSOR loop is dominated by the wavefront subroutines and therefore these have been the focus of our GPU optimisation strategy.

In the CPU implementation of LU, each call to `blts` or `buts` processes a single tile of grid-points with size $N_x/n \times N_y/m \times 1$, stepping through j and i in turn for some fixed value of k. On the other hand, the GPU implementation is closer to Lamport's original description of the Hyperplane algorithm [13] due to the SIMD nature of the architecture; the values of all grid-points on a given *hyperplane* defined by $i+j+k=f$ can be computed in parallel. Furthermore, our implementation is capable of processing grid-points in blocks of arbitrary depth (*i.e.* $N_x/n \times N_y/m \times h$). Setting h to a value greater than 1 (typically around $N_x/n+N_y/n-1$) has two advantages: (1) the number of MPI messages sent, as well as PCIe transfers for MPI communications, in each iteration decreases, at the expense of sending larger messages (which makes more efficient use of bandwidth) and (2) the GPU implementation can benefit from the increased level of parallelism afforded by three-dimensional wavefronts. However, setting h to too high a value causes delay to downstream processors which are waiting on data. Due to this performance trade-off, it is necessary to identify the opti-

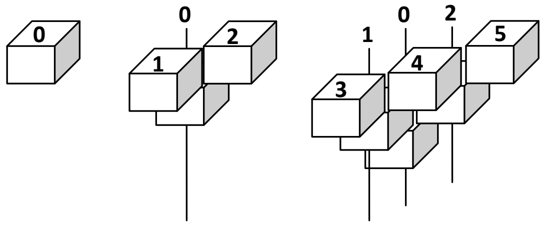


Figure 1: Two-dimensional mapping of threads in a thread-block on to the entire three-dimensional grid.

Table 1: Specification of the CUDA-capable GPUs used.

	GeForce		**Tesla**		
	8400GS	**9800GT**	**T10**	**C1060**	**C2050**
Cores	8	112	240	240	448
Clock Rate (GHz)	1.40	1.38	1.30	1.30	1.15
Global Memory (GB)	0.25	1	4	4	3[*]
Shared Memory (per SP)	16kB	16kB	16kB	16kB	16kB[†]
Compute Cap.	1.1	1.1	1.3	1.3	2.0

mal value of h through empirical evaluation or performance modelling.

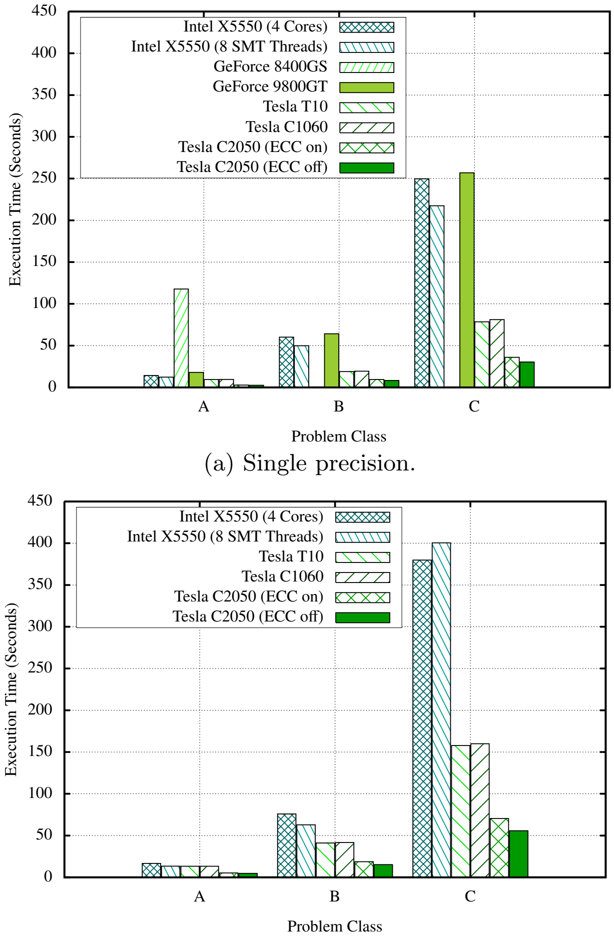
Global synchronisation is provided by repeated kernel invocations, with each kernel launch solving a separate hyperplane. As shown in Figure 1, which details the mapping of threads to grid-points in a wavefront, the first kernel is responsible for computing the value of one grid-point, the second for three, the third for six and so on.

A further optimisation is related to the four arrays populated in the `jacld` and `jacu` subroutines. In the original Fortran benchmark, each of these subroutines compute 100 values per grid-point based on 20 array values. These 100 values are then read in by the `blts` and `buts` subroutines respectively, where they are used in relatively few calculations. On a GPU, the slower speed of access to global memory makes this form of access particularly expensive.

By reimplementing the `jacld` and `jacu` subroutines as four templated functions (with 25 template options each), the `blts` and `buts` subroutines need only retrieve the original 20 array values from memory, before computing the 100 values at the point in code where they are required. This has an additional benefit to the GPU implementation, in that it removes the need to store these values completely; the total amount of memory required for any given problem size decreases considerably, counter-balancing some of the excess memory used by our rearrangement kernels.

5. SINGLE WORKSTATION PERFORMANCE

Our first set of experiments is directed at comparing the performance of the GPU solution on a single workstation. Problem classes A, B and C of the benchmark were executed in both single and double precision on single workstations with a range of NVIDIA GPUs. These include two consumer-grade low-end GPUs (GeForce 8400GS and GeForce 9800GT) and three HPC-capable GPUs (Tesla T10, Tesla C1060 and the Tesla C2050 based on the new "Fermi" architecture). The CPUs used in all workstations (with the exception of the T10 workstation) are "Nehalem"-class 2.66GHz Intel Xeon X5550s, with 12GB of RAM and the ability to utilise dual-issue simultaneous multi-threading (SMT); the CPU in the T10 workstation is a "Nehalem"-class 2.53GHz Intel Xeon E5540; the hardware specifications of the different GPUs can be found in Table 1. To ease reproducability, the compiler toolkits and options for each platform are listed in Table 2. The graphs in Figures 2(a) and 2(b) show the resulting execution times.



(a) Single precision.

(b) Double precision.

Figure 2: Execution times across different workstation configurations (CPU and GPU).

As shown in Table 1, the GeForce 8400GS and 9800GT are both of compute capability 1.1. As a result, they are unable to support double precision calculations and are therefore not included in Figure 2(b). Furthermore, the 8400GS has only 256MB of global memory available to it, limiting it to the processing of Class A problems at maximum.

As shown in the graphs, the GPU solution outperforms the original Fortran benchmark for all three problem classes; the Tesla T10 and C2050 are up to 2.8x and 7.7x faster in

[*] 2.65GB with ECC-enabled

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

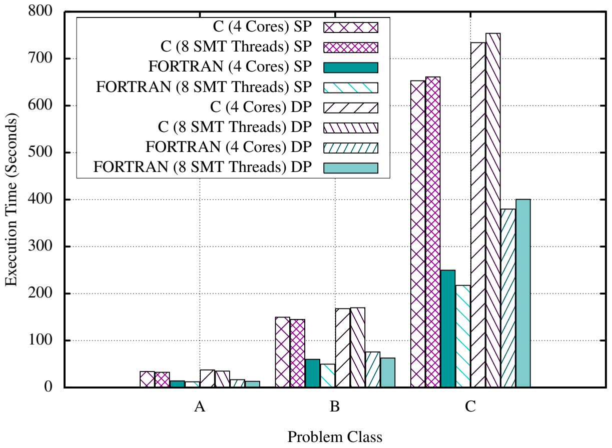


Figure 3: Fortran and C (without CUDA) Runtimes

Table 2: Platform and Compiler Details

Device	Compiler	Options

C1060/T10 (Host)	GNU 4.3	`-O2 -msse3`
		`-funroll-loops`

C1060/T10 (GPU)	NVCC	`-O2 -arch="sm_13"`

C2050 (Host)	GNU 4.3	`-O2 -msse3`
		`-funroll-loops`

C2050 (GPU)	NVCC	`-O2 -arch="sm_13"`

Nehalem	Sun Studio 12
	(Update 1)
		`-O5 -native`
		`-xprefetch -xunroll=8`
		`-xipo -xvector`

single precision and up to 2.3x and 6.9x faster in double precision respectively. The most surprising result is that the performance of the GeForce 9800GT is very close to that of the Nehalem for single precision. We suspect that a newer consumer-grade card of higher compute capability (*e.g.* a GTX280 or GTX480), would perform better still, making a compelling argument for the utilisation of GPUs for the acceleration of codes in single-workstation environments.

Figure 3 presents the runtime of our C port prior to the use of CUDA (*i.e.* all mathematics are ported and running on only the CPU). Note that the port to the C language was structured from the outset to be more amenable to CUDA rather than being optimised for CPU execution. The effect of this is that when executed on the CPU the runtime almost doubles in comparison to the Fortran version supplied by NASA, demonstrating that the performance improvements shown in Figures 2(a) and 2(b) are from the use of the GPU rather than from any optimisations introduced during the port to C.

Where applicable, the performance difference between single and double precision on GPUs is significant. Across all three of the GPUs capable of double precision arithmetic (*i.e.* Tesla T10, C1060 and C2050) there is a speedup of at least 2x from using single precision. The performance difference for CPUs is less significant, with the double precision version of the code being 1.11x and 1.35x slower for the Class B and C problems respectively.

Another notable difference is the performance of each of the GPUs. The execution times of the Tesla T10 and C1060 are consistently around 3x faster than those of the 9800GT, whilst those of the Tesla C2050 are around 2.2x faster still. This clearly illustrates the improvements that NVIDIA have made to the CUDA architecture between devices of different compute capability. For example, in our experiments the Tesla C2050 is up to 8.5x faster than the 9800GT, despite the fact that it only has 4x the number of cores – and that the clock speed of those cores has been decreased from 1.38Ghz to 1.15GHz.

Table 3: Predictions vs. Benchmarked Times for the Class-C Problem

Machine	Count	Predicted	Benchmarked
Tesla C1060	1	153.79	157.71
	4	71.59	70.22
	16	46.14	46.28
BlueGene/P	128	58.80	58.95
	256	32.59	33.14
	512	19.67	20.25

Table 4: LU Class-E based like-for-like time to solution

	Device Count	Projected Runtime (secs)	Max. Power (kW)	Theo. Peak (TFLOP/s)
Tesla C2050	256	233.33	59.5	131.9
BlueGene/P	8192	217.26	32.8	27.9
Tesla C2050	4096	143.26	974.8	2110.3
BlueGene/P	16384	118.29	65.6	55.7

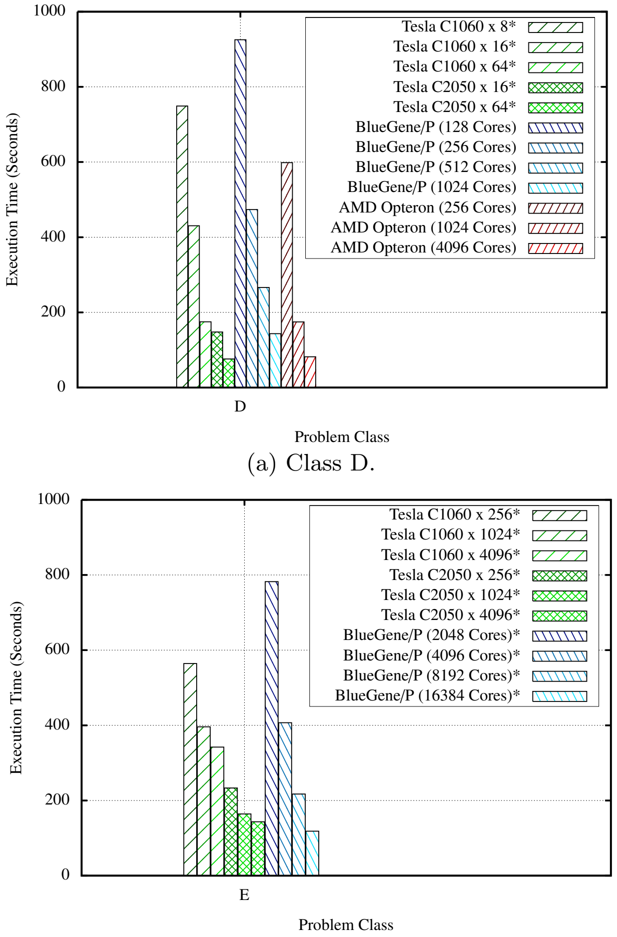
One of the most eagerly anticipated additions to NVIDIA's Fermi architecture was the inclusion of ECC memory. The Tesla C2050's ECC memory is not without cost, however; firstly, and as shown in Table 1, enabling ECC decreases the amount of global memory available to the user from 3GB to 2.65GB; secondly, enabling ECC leads to a not insignificant performance decrease. For a Class C problem run in double precision, execution times are almost 1.26x lower when ECC is disabled.

Though these results illustrate the performance benefits of GPU utilisation at single workstation level, the question of whether GPUs should be adopted within distributed MPI-based clusters remains unanswered. In the following section, we attempt to answer this open question and compare the performance of our GPU solution running at scale to the performance of a number of production-grade HPC CPU clusters.

6. PERFORMANCE AT SCALE

Figures 4(a) and 4(b) present execution times for the Class D and E problems respectively when run at scale. These times were predicted using a recently developed plug-and-play model of wavefront computations [16] or, where possible, measured using the Fortran benchmark. The CPUs featured in these graphs are the quad-core PowerPC 450d (BlueGene/P) which runs at 850MHz and the quad-socket, quad-core AMD Opteron 8356 which runs at 2.3GHz.

For GPU predictions the model has been altered so that the computation times align with the algorithmic changes discussed in Section 4 and the MPI communication times include the PCIe transfer times associated with reading data from the GPU device. The PCIe latencies and bandwidths for both GPU cards (Tesla C1060 and C2050) were obtained via the "bandwidthTest" benchmark provided in the NVIDIA CUDA SDK. The MPI communication times are based on an enlarged version of a previously benchmarked



(a) Class D.

(b) Class E.

Figure 4: Execution times for large scale GPU and CPU clusters (* indicates a model-based prediction).

InfiniBand system. This GPU model validates with 95% accuracy for Class C runs at 1, 4 and 16 GPU devices (see Table 3) and with similar accuracy on the BlueGene/P system between 128 and 512-cores.

These results demonstrate the competitive abilities of a GPU solution with the C2050 processors providing approximately equal runtime to the BlueGene/P system at between 4 and 32 times fewer nodes (for the Class E problem). We note that whilst this seems to favour the GPU in terms of pure runtime, the power required by the C2050 GPU devices (up to 230 Watts per device) exceeds that of the BlueGene/P system, which requires approximately 16 Watts per quad-core processor card [19].

The reader's attention is also drawn to the higher levels of scalability found in the BlueGene/P system, which is a result of the system's integrated design. The GPU solution provides lower scalability because of the declining levels of in-device parallelism which result from the use of strong-scaling. We expect that as the number of execution threads permissible in a GPU solution increase with forthcoming designs, there will be significant pressure on the designs of machines such as BlueGene to provide even higher levels of scalability in order to remain competitive. We also note the higher levels of efficiency of the BlueGene/P solution in which 16,384 cores have a theoretical peak performance of 55 TFLOP/s compared to, for example, a 256 C2050 solution which has a theoretical peak double-precision performance of 131 TFLOP/s. This identifies a potential issue with the construction of future machines; GPU-based solutions will no doubt achieve high placements in rankings such as the Top-500, but may deliver lower sustained performance than highly integrated solutions (such as the BlueGene). Table 4 presents a summarised like-for-like comparison in which we fix the time-to-solution, and compare the competing GPUs and CPU processors required to deliver these results.

7. CONCLUSIONS

The past two to three years have seen significant interest in the adoption of general-purpose GPUs for solving large scale scientific problems. In this paper, we have presented a port of the NAS-LU benchmark to GPUs, using a hybrid of MPI and NVIDIA's CUDA, and show runtimes for a variety of consumer and high-end HPC NVIDIA GPUs as well as processors from leading HPC manufacturers including Intel, AMD and IBM.

At the workstation level, the results show that the C1060 and C2050 GPUs from NVIDIA are between 3x and 7x faster than a high-end quad-core Nehalem processor solving the same Class C problem. At scale, we compare the performance of our solution running across multiple GPU nodes with an InfiniBand-based cluster of AMD processors and an IBM BlueGene/P solution, augmenting known runtimes with projections from a recently developed analytical model of LU. For the Class E LU problem – currently the largest class in the NAS benchmark suite – 256 Tesla C2050 GPUs provide approximately equivalent performance to 8,192 BlueGene/P cores (32x on a GPU-to-core basis). Whilst this level of raw performance is competitive, often exceeding many general-purpose processor offerings in a device-to-device comparison, the power-efficiency of the GPU solution is lower than that of the BlueGene/P solution, highlighting the lower levels of sustained performance currently realisable from a GPU solution.

These performance results raise interesting questions about the future direction of HPC architectures. On the one hand, we might expect to see smaller clusters of SIMT or GPU-based solutions 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. The results in this paper give some indication of the level of performance which might be expected from these two alternatives for a genuinely representative benchmark code.

Acknowledgements

Access to the NVIDIA workstations benchmarked in this paper is supported by the Royal Society Industrial Fellowship of Prof. Stephen Jarvis. We are grateful to Scott Futral, Jan Nunes and the Livermore Computing Team for access to, and help in using, the DawnDev BlueGene/P machine located at the Lawrence Livermore National Laboratory. The authors would also like to thank the High Performance Com-

puting team at the Daresbury Laboratory (UK) for access to the Daresbury multi-card GPU-cluster.

8. REFERENCES

[1] CUDA Community Showcase. http://www.nvidia.com/object/cuda_showcase_html.html, 2010.

[2] Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V., and Weeratunga, S. The NAS Parallel Benchmarks. *International Journal of High Performance Computing Applications 5*, 3 (1991), 63–73.

[3] Barszcz, E., Fatoohi, R. A., Venkatakrishnan, V., and Weeratunga, S. K. Solution of Regular, Sparse Triangular Linear Systems on Vector and Distributed-Memory Multiprocessors. Tech. Rep. RNR-93-007, NAS Applied Research Branch, NASA Ames Research Center, Moffett Field, CA 94035, April 1993.

[4] Bordawekar, R., Bondhugula, U., and Rao, R. Believe it or Not! Multi-core CPUs Can Match GPU Performance for FLOP-intensive Application! Tech. Rep. RC24982, IBM, 2010.

[5] Gong, C., Liu, J., Gong, Z., Qin, J., and Xie, J. Optimizing Sweep3d for Graphic Processor Unit. *Algorithms and Architectures for Parallel Processing 6081* (2010), 416–426.

[6] Govindaraju, N. K., Lloyd, B., Dotsenko, Y., Smith, B., and Manferdelli, J. High Performance Discrete Fourier Transforms on Graphics Processors. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (Piscataway, NJ, USA, 2008), IEEE Press, pp. 1–12.

[7] Gress, A., and Zachmann, G. GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures. In *Proc. 20th IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS 2006)* (Rhodes Island, Greece, 25–29 2006).

[8] Harish, P., and Narayanan, P. J. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *IEEE International Conference on High Performance Computing* (December 2007), IEEE Computer Society.

[9] Herruzo, E., Ruiz, G., Benavides, J. I., and Plata, O. A New Parallel Sorting Algorithm based on Odd-Even Mergesort. In *PDP '07: Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 18–22.

[10] Hoisie, A., Lubeck, O., Wasserman, H., Petrini, F., and Alme, H. A General Predictive Performance Model for Wavefront Algorithms on Clusters of SMPs. In *Proceedings of the 2000 International Conference on Parallel Processing* (2000), IEEE Computer Society, p. 219.

[11] Jacobsen, D. A., Thibault, J. C., and Senocak, I. An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters. In *48th AIAA Aerospace Sciences Meeting and Exhibit* (Orlando, FL., Jan 2010).

[12] Katz, G. J., and Kider, Jr, J. T. All-Pairs Shortest-Paths for Large Graphs on the GPU. In *GH '08: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware* (Aire-la-Ville, Switzerland, Switzerland, 2008), Eurographics Association, pp. 47–55.

[13] Lamport, L. The Parallel Execution of DO Loops. *Commun. ACM 17*, 2 (1974), 83–93.

[14] Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., and Dubey, P. Debunking the 100x GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. *SIGARCH Comput. Archit. News 38*, 3 (2010), 451–460.

[15] Ltaief, H., Tomov, S., Nath, R., Du, P., and Dongarra, J. A Scalable High Performant Cholesky Factorization for Multicore with GPU Accelerators. Tech. Rep. UT-CS-09-646, University of Tennessee, Knoxville, November 2009.

[16] Mudalige, G. R., Vernon, M. K., and Jarvis, S. A. A Plug-and-Play Model for Evaluating Wavefront Computations on Parallel Architectures. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008)* (April, 2008), IEEE Computer Society.

[17] Munekawa, Y., Ino, F., and Hagihara, K. Design and Implementation of the Smith-Waterman Algorithm on the CUDA-Compatible GPU. In *8th IEEE International Conference on Bioinformatics and Bioengineering (BIBE 2008)* (October 2008), IEEE Computer Society.

[18] Nukada, A., Ogata, Y., Endo, T., and Matsuoka, S. Bandwidth Intensive 3-D FFT kernel for GPUs using CUDA. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (Piscataway, NJ, USA, 2008), IEEE Press, pp. 1–11.

[19] Pedretti, K., Kelly, S., and Levenhagen, M. Summary of Multi-Core Hardware and Programming Model Investigations. Tech. Rep. SAND2008-3205, Sandia National Laboratory, Albuquerque, New Mexico, USA, May 2008.

[20] Petrini, F., Fossum, G., Fernandez, J., Varbanescu, A. L., Kistler, M., and Perrone, M. Multicore Surprises: Lessons Learned from Optimizing Sweep3D on the Cell Broadband Engine. In *International Parallel and Distributed Processing Symposium (IPDPS 2007)* (July 2007), IEEE Computer Society, pp. 1–10.

[21] Rodrigues, C. I., Hardy, D. J., Stone, J. E., Schulten, K., and Hwu, W.-M. W. GPU Acceleration of Cutoff Pair Potentials for Molecular Modeling Applications. In *CF '08: Proceedings of the 5th Conference on Computing Frontiers* (New York, NY, USA, 2008), ACM, pp. 273–282.

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

[23] Yarrow, M., and der Wijngaart, R. V. Communication Improvement for the LU NAS Parallel Benchmark: A Model for Efficient Parallel Relaxation

Schemes. Tech. Rep. NAS- 97-032, NASA Ames Research Center, November 1997.

[24] Zhang, Y., Cohen, J., and Owens, J. D. Fast Tridiagonal Solvers on the GPU. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2010), ACM, pp. 127–136.