An Investigation of the Performance Portability of OpenCL

S.J. Pennycook[a], S.D. Hammond[b], S.A. Wright[a], J.A. Herdman[c], I. Miller[c], S.A. Jarvis[a]

[a]*Performance Computing and Visualisation*
Department of Computer Science
University of Warwick, CV4 7AL, UK

[b]*Scalable Computer Architectures*
Center for Computing Research
Sandia National Laboratories
Albuquerque, NM 87185, USA

[c]*Supercomputing Solution Centre*
UK Atomic Weapons Establishment
Aldermaston, RG7 4PR, UK

Abstract

This paper reports on the development of an MPI/OpenCL implementation of LU, an application-level benchmark from the NAS Parallel Benchmark Suite. An account of the design decisions addressed during the development of this code is presented, demonstrating the importance of memory arrangement and work-item/work-group distribution strategies when applications are deployed on different device types. The resulting platform-agnostic, single source application is benchmarked on a number of different architectures, and is shown to be 1.3–1.5x slower than native FORTRAN or CUDA implementations on a single node and 1.3–3.1x slower on multiple nodes. We also explore the potential performance gains of OpenCL's device fissioning capability, demonstrating up to a 3x speed-up over our original OpenCL implementation.

Keywords: Many-Core Computing, GPU Computing, Optimisation, OpenCL, High Performance Computing

1. Introduction

Approximately twenty years ago, the High Performance Computing (HPC) industry underwent arguably its most significant technological shift to date. During this time the widespread use of vector-processor-based supercomputers gradually declined in favour of larger scale distributed-memory architectures.

Email address: `sjp@dcs.warwick.ac.uk` (S.J. Pennycook)

The success of distributed machines was their ability to offer superior price-performance over their complex vector counterparts. This drastic change in computing hardware produced an equally significant switch in the design and engineering of parallel scientific applications. Where parallelism was previously created through the identification of vectorisation opportunities in loops, parallelism in the distributed-memory paradigm required a complete redesign of many algorithms to utilise explicit communications through the Message Passing Interface (MPI) communications library.

As the computing community begins to address the utilisation of supercomputers that are able to calculate 10^{15} floating point operations per second (*i.e.* one petaflop/s), we see another potential paradigm shift in the construction of parallel computing hardware and software. While alternative architectural designs exist, the use of *computational accelerators* appears to be gaining significant traction in academia and industry alike. In these designs, specialised hardware devices such as graphics processing units (GPUs), dense multi-core processors and field programmable gate arrays (FPGAs) are utilised alongside conventional general-purpose processors to accelerate specially selected sections of code. The use of accelerator-based architectures appears promising due to the significant levels of raw performance that can be achieved and a high performance-to-power cost ratio, reflecting some of the same motivations for distributed computing at the end of the 1990s.

From an application design and porting point of view, the adoption of distributed computational accelerators presents several challenges beyond constructing the initial message-passing structure of an application. These challenges include (*i*) how and where to locate program data, since many accelerator devices have localised storage (*data locality*); (*ii*) how to structure data to improve performance (*memory layout*); (*iii*) how to efficiently transfer data between the main general-purpose processor and any accelerator devices (*data transfer cost*); and (*iv*) how to develop an application such that it can run across different hardware architectures and varieties of accelerator (*portability*). This last point in particular has proven to be a major technological challenge, since each new hardware offering has traditionally required applications to be re-engineered and re-tuned with a new programming toolkit (*e.g.* CUDA, Brook+, Cray/PGI accelerator directives).

The Open Computing Language (OpenCL [1]) is a recently developed cross-vendor standard which ensures the *functional* portability of codes between hardware from a number of vendors, thereby eliminating the need for applications to be re-coded on a per-device or per-programming toolkit basis. However, it makes no guarantees of *performance* portability – if an OpenCL application is too highly tuned to a particular architecture, it is likely to exhibit very different levels of performance on others. While this is perhaps more desirable than having to re-write code for each new platform, the effect of poor performance portability may provide a barrier to the re-use of application code across devices.

The purpose of the work presented in this paper is to assess whether it is possible to maintain a single application written in OpenCL that is able to achieve *acceptable* performance across a variety of different platforms. We note that this dictates that extensive platform-specific optimisations (*e.g.* tuning for particular instruction sets or hardware features) has *not* been carried out. Instead, we attempt to provide a more generic software development approach that maintains performance portability, at the cost of some performance compromise compared to native (hand-tuned) implementations. We acknowledge that we do use auto-tuning on two important parameters (memory layout and work-item distribution) to improve performance, but these do not affect the underlying algorithm.

We illustrate our study by developing an entirely new OpenCL-based implementation of the LU benchmark, which originates from the NAS Parallel Benchmark (NPB) Suite [2]. This builds on our experience in developing a version of this benchmark for NVIDIA's CUDA architecture, work which is described in [3, 4], and extends the application to a potentially broader range of hardware.

The specific contributions of this paper are:

1. We present the first reported MPI/OpenCL implementation of an application level benchmark from the NPB suite, with an account of the design decisions addressed during its development. We note that the porting of LU is interesting for two reasons – first, LU is recognised as an *application-class* benchmark because of its size and complexity; and second, the mathematics implemented by LU (LU-factorisation and Successive Over-Relaxation) are common to a wide variety of complex scientific applications;

2. Using industry leading CPU and GPU devices from AMD, Intel and NVIDIA, we demonstrate the importance of memory arrangement and work-item/work-group distribution strategies for each device type. We also compare the performance of our OpenCL implementation of LU with a native FORTRAN 77 implementation on a CPU, and a CUDA implementation on GPUs. Two different OpenCL software development kits (SDKs) are employed (AMD and Intel) and two different FORTRAN 77 compilers are used (GNU and Sun Studio), to ensure a fair comparison;

3. We extend the study to examine multi-node performance. We do this using two systems: (*i*) a cluster of dual-socket, hex-core Intel X5660 nodes; and (*ii*) a cluster of nodes containing Intel X5650s and NVIDIA M2050s. Two algorithmic optimisations related to "k-blocking" are implemented in LU, and are shown to significantly improve the performance of the OpenCL code;

4. We examine the potential performance gains of *device fission*, by which OpenCL is able to "fission" (*i.e.* split) a single device into multiple sub-devices at runtime. This gives a program greater freedom in assigning work to specific cores, allowing for task-level parallelism and/or better exploitation of temporal and spatial cache locality.

The remainder of this paper is organised as follows: Section 2 discusses related work; Section 3 outlines the operation of LU (and other wavefront applications) and provides an overview of the OpenCL programming model; Section 4 describes the development of a single-source OpenCL implementation of LU, which is then benchmarked on single compute nodes in Section 5 and across multiple compute nodes in Section 6; device fissioning is explored in Section 7 and we discuss our findings in Section 8; Section 9 concludes the paper.

2. Related Work

The achievable performance of OpenCL applications has been the subject of previous work. This has compared the performance of OpenCL codes with alternative platform-specific programming languages and methodologies (*e.g.* CUDA, Brook+, MPI) or has investigated the overheads associated with running OpenCL codes optimised for one architecture on a number of other architectures. Other work has investigated the use of OpenCL on different platforms as a means of assessing power usage [5] and productivity [6].

In [7] the authors investigate the use of OpenCL for the GPU acceleration of two BLAS functions. OpenCL implementations are compared to CUDA on NVIDIA GPUs and to ATI's Intermediate Language (IL) on an ATI Radeon 5870, and are shown to run significantly slower in both cases. The authors demonstrate that the performance of kernels tuned for NVIDIA GPUs is poor on ATI GPUs and vice-versa, concluding that an auto-tuning approach (which selects the appropriately tuned kernel at run-time) will provide performance portability for OpenCL applications. As the work is concerned primarily with GPU architectures, the overheads of running OpenCL kernels on CPU architectures is not considered.

A similar study [8] investigates the performance of several OpenCL applications on a number of different compute devices. Through examination of PTX code, the authors demonstrate that an initially large performance gap between CUDA and OpenCL is the result of compiler immaturity and that, following hand-optimisation (*e.g.* loop unrolling), OpenCL performance can match that of CUDA. Although performance results for CPUs are given, they are 20–183x slower than the equivalent GPU results, suggesting that the OpenCL kernels have been highly tuned for GPU architectures. The authors propose an investigation of auto-tuning as a method of achieving performance portability as future work.

The work of Weber et al. [9] is most similar to our own, providing a performance comparison of AMD and NVIDIA GPUs, multi-core CPUs and an FPGA using an OpenCL implementation of a Quantum Monte Carlo application. At large enough problem sizes, the performance of OpenCL is shown to exceed that of a native C++ code executing on a quad-core Intel CPU. However, the C++ code was compiled with the GCC compiler and `-O3`, the performance of which is not necessarily representative of a code compiled with a proprietary or industrially recognised compiler (*e.g.* Intel, PGI, Sun) at higher optimisation levels.

The work found in this paper is also concerned with the performance portability of OpenCL. We are not just interested in performance *per se*, but in programming techniques and algorithmic optimisations which allow application scientists to advance their code in a predictable fashion, despite architectural changes to the platforms on which they run. The papers described in this section have tried to achieve performance portability by maintaining individually tuned OpenCL kernels for each platform; we seek *acceptable* performance across platforms, but intentionally avoid architecture-specific optimisations (that may ensure the *best* performance) which do not translate to other platforms.

3. Background

3.1. Wavefront Applications and the LU Benchmark

LU is an application-level benchmark from the NPB suite, a set of parallel aerodynamic simulation benchmarks designed by domain-scientists at NASA to assess the performance of representative calculations on large-scale supercomputers. 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. A thorough discussion of the mathematics employed can be found in [2].

The three-dimensional data grid used by LU is of size N^3 (*i.e.* the problem space is always cubic), although the underlying wavefront algorithm works equally well on grids of any dimension. As of release 3.3.1, NASA provide seven different application "classes" for which the benchmark is capable of performing verification. We focus on three of these classes in this work: Class A (64^3), Class B (102^3) and Class C (162^3). The use of these problem sizes ensures results which successfully validate and provide direct performance comparisons to those reported elsewhere.

In the MPI implementation of LU, 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 (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, while the original processor solves its second tile. Once the neighbouring processors have completed their tiles, their edge data is sent downstream. This continues until the processor at the opposite vertex to the starting processor solves its last 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 other parallel applications including the Sweep3D [10] and Chimaera [11] particle transport applications. 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 in the US and the Atomic Weapons Establishment (AWE) in the UK. The SSOR-preconditioning and LU-decomposition methods utilised by the LU application are also routinely employed in parallel solvers throughout many scientific domains, making the implementation described here of broad interest.

LU executes two sweeps through the data grid (as opposed to eight in applications such as Sweep3D), one originating from the vertex held by processor 0, and another in the opposite direction (which finishes at processor 0). Although the execution time for a single iteration of LU is small, typically taking minutes, when scaled to larger problems and incorporated as part of a larger workflow requiring hundreds or thousands of iterations prior to convergence, the application can consume vast amounts of processing time. The principal use of the LU benchmark is comparing the suitability of different architectures for production CFD applications [12] and thus the results presented in this paper have implications for large-scale production codes. The memory requirements of LU are also significant ($\approx$ 160 GB for a 1020^3 Class E problem) – the amount of RAM available per node in commodity clusters is typically much less than this, and the memory available to many current accelerator architectures is limited by their use of GDDR, thus necessitating the use of large distributed machines.

The LU benchmark was selected for this case-study because it represents a difficult problem to parallelise effectively. Previous work that presents the parallelisation of wavefront applications for novel architectures [13, 14, 15, 16, 17], and our own previous efforts in porting this specific benchmark to CUDA [3], have shown that good performance for a complete application can be achieved for this type of algorithm on GPUs. However, previous implementations have all focused on a single architecture and typically employ a different parallelisation approach to that used in CPU implementations.

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
```

The pseudocode in Algorithm 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: `jacld` and `jacu` carry out a number of independent computations per grid-point, which can be executed in parallel, to pre-compute the values of arrays used in the forward and backward wavefront sweeps; `blts` and `buts` are responsible for the forward and backward sweeps respectively; `l2norm` computes a parallel reduction (on user-specified iterations); and `rhs` carries out three parallel stencil update operations, which have no data dependencies between grid-points. The number of loop iterations is configurable at both compile- and run-time, but is typically 250–300.

3.2. The OpenCL Programming Model

The OpenCL programming model is conceptually similar to that of NVIDIA's CUDA, except that it is capable of targeting a range of different hardware platforms from a number of different vendors. The language specification is contributed to by many industry vendors and institutions, but maintained by the Khronos Group [1].

With OpenCL, a *host* (typically a CPU) runs C/C++ code using library calls to communicate with and control one or more *devices*. Each device is made up of a number of *compute units* and may be a GPU, an accelerator, a multi-core CPU and/or the host itself. Compute units can be further sub-divided into *processing elements*.

Functions run on a device are known as *kernels* and can be compiled just-in-time (JIT) from source, or loaded from a cached binary if one exists for the current target platform. These kernels are executed in a single-program-multiple-data (SPMD) fashion by a one-, two- or three-dimensional set of *work-items*, which are grouped together into *work-groups*. Some architectures may choose to execute work-items in a single-instruction-multiple-data (SIMD) fashion.

In order to permit OpenCL applications to scale up or down to fit different hardware configurations, work-groups can be scheduled for execution to any available compute unit and in any order. This means that, unless carefully designed, programs that rely on synchronisation across all work-items are likely to deadlock. Indeed, the OpenCL programming model does not currently provide any method of global synchronisation for this reason. There is, however, an implicit global synchronisation barrier between separate kernel calls and local synchronisation is possible between all work-items in a given work-group.

Before addressing the performance of OpenCL – including cross-platform optimisation – it is important to understand how the OpenCL concepts of *devices*, *compute units*, *work-groups* and *work-items* map to different architectures (and sometimes to the same architecture, when there may be more than one implementation of the OpenCL runtime available).

On multi-core CPUs, we use version 2.4 of the Accelerated Parallel Processing (APP) SDK from AMD [18], which is compatible with any AMD or Intel CPU that supports SSE instructions. All of the CPUs in a node are treated as a single device, with each of the cores presenting themselves as compute

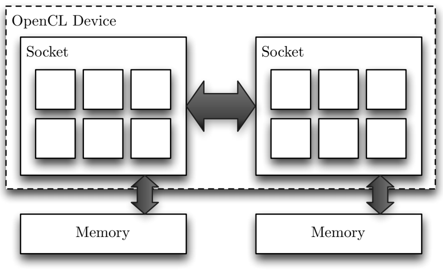


Figure 1: A dual-socket, hex-core node as it appears to the OpenCL runtime.

Device	**Compute Units**	**Processing Elements**
Intel X5550, 2.66 GHz	4	4
Intel X5660, 2.80 GHz	12	12
NVIDIA Tesla C1060	30	240
NVIDIA Tesla C2050	14	448
AMD/ATI FirePro V7800	18	1440

Table 1: Devices as they appear to the OpenCL runtime.

units (see Figure 1). Each work-group is scheduled to a hardware thread and is executed on a single core, with the chosen core iterating (in order) over all work-items in the work-group. Use of OpenCL's vector types (*e.g.* `float4`, `double2`) is required to utilise the SSE units of each core. We also make use of Intel's OpenCL implementation [19], the behaviour of which is very similar; the key difference is that Intel's implementation supports auto-vectorisation in addition to OpenCL vector types (where possible, the compiler attempts to pack the work of contiguous work-items into SSE vector units automatically).

On NVIDIA GPUs, we use version 4.0 of the CUDA Toolkit and SDK [20], which includes OpenCL support. Work-groups map to CUDA blocks, which are scheduled to the GPU's stream-multiprocessors (SMs), while work-items map to CUDA threads and are executed synchronously by streaming processors (SPs) in sets of 32 known as *warps*. The approach is similar for AMD/ATI GPUs, except that we use the AMD SDK; work-groups are scheduled to the GPU's SIMD units, while work-items are executed synchronously by streaming cores in sets of 64 known as *wavefronts* (not to be confused with the wavefronts in LU). As the GPUs from both vendors are separate devices to the host CPU, all data to be used by kernels must be transferred to the device via PCI-Express (PCIe).

Table 1 lists the number of compute units and processing elements for each of the OpenCL devices used in our experiments. It should be noted that the meaning of the term "processing element" differs by architecture; on CPUs, a traditional x86 core is a compute unit of one processing element; on NVIDIA GPUs, each stream multiprocessor (compute unit) consists of 8 or 32 "CUDA cores" (processing elements) on the Tesla and Fermi architectures respectively; and on AMD/ATI GPUs, a compute unit contains 16 stream cores of 5 simple processing elements each.

4. Implementation

4.1. Motivations for a Single Source

In recent years the HPC industry has shown a great deal of interest in the use of novel architectures. This has been partly driven by the gradual maturity of software environments for such devices but also because of the significant levels of performance such devices can offer. One of the biggest challenges in adopting these devices is the porting effort that is required, particularly in the context of legacy applications. In many cases the performance of such applications has been optimised for specific types of processor or machine architecture, introducing additional complexity into the code which will not easily port to accelerator devices. In such cases there is no clear path to code development.

There are several reasons therefore to assess the practicality of a single source approach to application design: (*i*) it is easier to maintain a single code that targets all platforms, as opposed to separate hand-tuned versions of the same code for each alternative platform; (*ii*) it reduces the risk of being locked into a single vendor solution; (*iii*) benchmarking is simplified, as the results can be compared from a single code source; and (*iv*) it represents a "safer" investment for HPC sites, as new codes (and ported legacy codes) will run on both existing and future architectures. OpenCL is not the only available option for maintaining a single-source application; source-to-source translation tools, algebra libraries with support for multiple device types (*e.g.* BLAS) and domain-specific languages are all viable alternatives.

4.2. Ensuring Performance Portability

One of the issues associated with even simple CUDA programs is that optimisation can be very difficult. The specifications of CUDA devices vary in several respects (*e.g.* number of registers per work-item, amount of shared memory, coalescence criteria) and each kernel has a number of adjustable parameters (*e.g.* the number of work-items and work-groups). As such, the optimal values for these parameters on one architecture may not be optimal on others. Several papers have suggested that this issue is best handled through "auto-tuning", a process that sees a code automatically searching a given parameter space as it runs and tuning itself to maximise platform performance, for CUDA and OpenCL codes alike [7, 8, 21]. We adopt this parameterisation approach in our OpenCL implementation of LU, focusing on two high-level criteria we believe to be important when targeting multiple platforms: *memory layout* and *work-item/work-group distribution*.

Although we acknowledge that the choice between scalar and vector data types has a large impact on the performance of some architectures, the parameterisation of this space requires much more involved low-level code and algorithm changes than memory layout and work-item distribution. The Intel

SDK already supports auto-vectorisation for CPUs, while the AMD SDK is capable of packing the VLIW architecture of an AMD GPU without the use of explicit vector types. We expect that the auto-vectorisation capabilities of these compilers will improve with time. Some effort is likely to be required from a programmer to ensure successful auto-vectorisation of their kernels, but such efforts are application-specific and beyond the scope of this paper.

The choice of floating-point precision is also an important parameter. Across CPUs and GPUs, double precision compute is approximately twice as slow as single precision, and the cost of data movement (*i.e.* memory copies or MPI/PCIe communication) is similarly affected. All of the results presented in this paper use double precision, in keeping with the precision of the original LU benchmark. Supporting multiple precisions within a single-source application could very easily be achieved through the use of macros and the C pre-processor.

4.3. Memory Layout

The performance of codes written for both CPUs and GPUs are sensitive to memory layout. Due to the multiple-instruction-multiple-data (MIMD) parallelism found in multi-core CPUs, codes where each core operates in its own separate "chunk" of memory are likely to outperform those where cores are required to copy data from one another's memory spaces. Furthermore, each core is likely to perform best when the memory layout makes good use of its cache line width through both spatial and temporal locality.

In terms of OpenCL, we would expect the best memory layout for CPUs (executing scalar code) to be one that allows each work-item to load in all of the data it will need in a single memory access. This memory layout is unlikely to be effective on GPUs, owing to the way in which work-items are grouped for synchronous execution in a native SIMD fashion. Both AMD and NVIDIA stress the importance of *coalescing* memory accesses (*i.e.* ensuring that contiguous work-items access contiguous memory locations) as this makes best use of memory bandwidth.

Our OpenCL implementation of LU therefore allows for the memory layout to be chosen at runtime. Currently, we support the two alternative layouts shown in Figure 2: array-of-structs (AoS) and struct-of-arrays (SoA). The AoS approach ensures that the five values associated with each LU grid-point are next to one another in memory, providing good cache utilisation for a scalar work-item; the SoA approach ensures that the five values are split into five separate units, allowing work-items processed in SIMD to access corresponding elements in parallel.

To support these alternative memory layouts in our implementation, the index of each array access is replaced with a macro or inline function (*i.e.* `array[k][j][i][m]` in the original C/C++ becomes `array[array_index(k, j, i, m)]`). This allows the index calculation for each array to be defined at kernel compile time. We also introduce kernel calls for rearranging memory based on these layouts immediately after a memory buffer is transferred to the device, or immediately before it is transferred back to the host. Our solution also rearranges memory between calls to the hyperplane and stencil operation

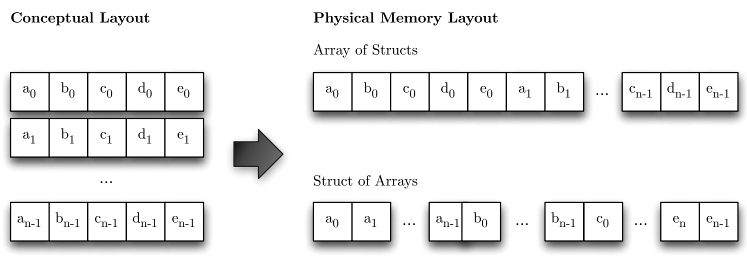

Conceptual Layout
Physical Memory Layout
Array of Structs
Struct of Arrays
a_0 b_0 c_0 d_0 e_0
a_1 b_1 c_1 d_1 e_1
...
a_{n-1} b_{n-1} c_{n-1} d_{n-1} e_{n-1}
a_0 b_0 c_0 d_0 e_0 a_1 b_1 ... c_{n-1} d_{n-1} e_{n-1}
a_0 a_1 ... a_{n-1} b_0 ... b_{n-1} c_0 ... e_n e_{n-1}

Figure 2: A comparison of the Array-of-Structs (AoS) and Struct-of-Arrays (SoA) memory layouts.

kernels of LU, due to their different memory access patterns (see implementation for more details [22]).

4.4. Work-Item and Work-Group Distribution

The number of work-items that can execute an instruction in parallel on GPU architectures is typically very high, as is the number required to effectively hide memory latency. On an NVIDIA Tesla C2050, an integer or single precision floating-point operation can be executed for two warps (64 work-items) in two clock cycles on each of its 14 SMs; further, the maximum number of warps that can be "active" (*i.e.* ready to be switched in, should another warp stall on a memory operation) on each SM is 48, leading to 1,536 active work-items per SM. On CPU architectures, there is significantly less parallelism available (for scalar code). On an Intel X5550 without SMT enabled, each core can only execute one thread at any given time; with SMT enabled, each core can execute the instructions of two threads (so long as they are in different pipelines) in an attempt to hide memory and instruction latencies.

Our OpenCL implementation of LU has two different methods of work-item and work-group distribution. The first, which we refer to as *fine-grained* distribution, launches one work-item for each LU grid-point value that must be computed (regardless of how many compute units and processing elements the device has); the second, which we refer to as *coarse-grained* distribution, launches one work-group per compute unit. In both cases, the size of the work-group is set based on the compute unit's SIMD width. For CPUs, the SIMD width is treated as one (despite SSE having a SIMD width of two for double precision) since, at the time of writing, the Intel compiler does not auto-vectorise our kernels. For GPUs, the SIMD width is treated as 64 – although this number is lower than is typically used on NVIDIA hardware, the size of our work-groups is limited by register constraints.

To facilitate a change in work-item and work-group distribution at runtime, each of our kernels is enclosed in a set of three nested loops, as shown in Algorithm 2. *imax*, *jmax* and *kmax* refer to the maximum grid-point coordinates considered by a given kernel in each dimension; `get_global_id` and `get_global_size` are two built-in OpenCL functions that return a work-item's

Algorithm 2 Parameterisation of work-item and work-group distribution.

```
for $k$ = get_global_id(2) to $kmax$ step get_global_size(2) do
    for $j$ = get_global_id(1) to $jmax$ step get_global_size(1) do
        for $i$ = get_global_id(0) to $imax$ step get_global_size(0) do

            // kernel body

        end for
    end for
end for
```

ID and the total number of work-items in each dimension respectively. These loops are structured in such a way that the kernel will execute for every grid point from $(0, 0, 0)$ to $(imax, jmax, kmax)$, regardless of the number or configuration of the work-items and work-groups launched. For example, we consider two extreme cases: a single work-group containing a single work-item will loop from 0 to the maximum in steps of 1; and a set of $imax \times jmax \times kmax$ work-groups containing a single work-item will execute the kernel for exactly one grid-point each.

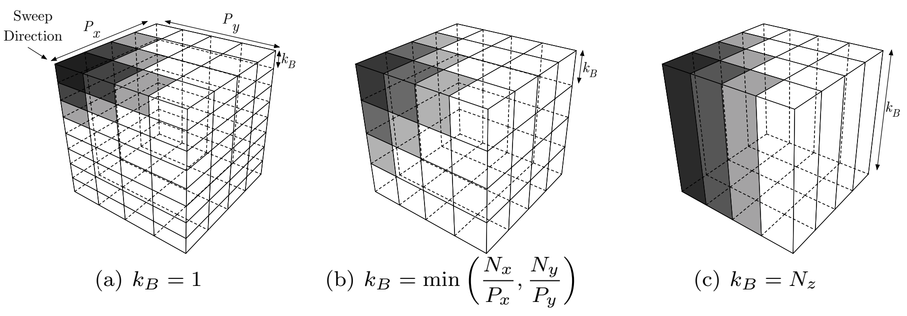
As mentioned previously, the number of work-items per work-group is identical for both the fine-grained and coarse-grained distribution strategies. It is the number of work-groups that changes, and hence the values returned by the `get_global_size` function. For fine-grained distribution, the function will return one of $imax$, $jmax$ or $kmax$, and thus each work-item will execute the kernel for the grid-point that corresponds to its ID. For coarse-grained distribution, each work-item will execute the kernel for $(imax \times jmax \times kmax)/(\text{SIMD width} \times \text{compute units})$ grid-points.

4.5. k-blocking

In the default version of LU, each processor solves a "tile" of size $N_x \times N_y \times 1$ at each time step prior to communication (see Section 3.1). Our OpenCL implementation employs an optimisation commonly known as k-blocking, a name which arises from previously reported studies on the Sweep3D wavefront code, which utilises axes labelled i, j and k. Rather than partitioning the z-axis into tiles of height 1, this optimisation sees each processor solve a block of depth k-block (or k_B) prior to communication.

The benefits of this optimisation are twofold. First, communication messages are aggregated into fewer, larger messages that make more effective use of network bandwidth at the potential cost of delaying downstream processors. Second, more parallelism is exposed within larger blocks. In a given wavefront step w, computation for a grid-point (i, j, k) can only proceed if $i + j + k = w$; for a "tile" with a fixed value of k, exploitable parallelism is restricted to the other two dimensions.

We must therefore choose a k-block that strikes a balance between providing a good level of parallelism on SIMD architectures and introducing significant



(a) $k_B = 1$ (b) $k_B = \min\left(\frac{N_x}{P_x}, \frac{N_y}{P_y}\right)$ (c) $k_B = N_z$

Figure 3: k-blocking policies.

delays to downstream processors. It is important to note that this performance trade-off is only a concern for multi-node runs; since a single node run contains no MPI communication, the k-block value can be set to maximise SIMD efficiency.

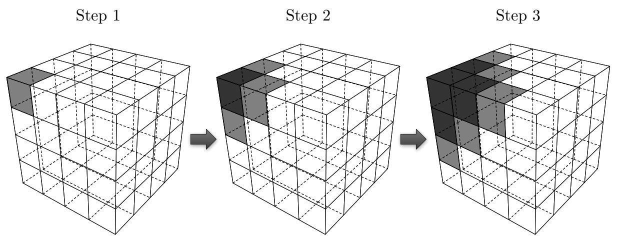
Figure 3 identifies three potential k-blocking depths. The current sweep-step is shown in light grey, downstream processors that are waiting for data are shown in white, and previous sweep steps are shown in progressively darker shades. A k-block depth of 1 (Figure 3(a)) minimises the amount of time any processor spends waiting on its first message, and represents the behaviour found in the original benchmark; a k-block depth of $\min(N_x/P_x, N_y/P_y)$ (Figure 3(b)) provides an approximately cubic unit of computation, thus balancing the need for a large k-block for compute efficiency with a small k-block for MPI efficiency; and a k-block of depth N_z (Figure 3(c)) maximises the surface of the 3D compute-face for as much of the run as possible.

4.6. k-block Compute Policy

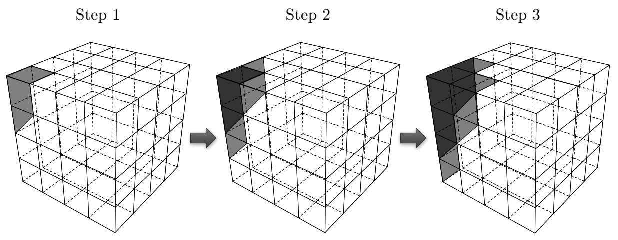
In [4] we presented an extension of the traditional k-blocking optimisation for CUDA devices. We extend this study to OpenCL-based executions on CPU and GPU devices here.

In a typical execution, the processing of a block of size $N_x/P_x \times N_y/P_y \times k_B$ will be conducted by executing a form of "mini-sweep" through the block. The size of the hyperplane $i + j + k = w$ starts at one data cell, increases until some maximum (dependent on the size of the block), and then decreases until it has passed through the last cell. Following communication, the next time step executes another "mini-sweep" for the next data block, again starting from a single cell. Conducting computation in this manner wastes parallel efficiency whenever the size of the hyperplane is less than the number of work-items an architecture can execute in parallel. Figure 4(a) demonstrates the effect of applying the traditional k-block compute policy in which the data is processed as a discrete chunk.

Our optimisation permits a processor to solve for grid-points beyond the current k-block boundary, thus maintaining the 45-degree angle established during



(a) Original Policy



(b) New Policy

Figure 4: Comparison of two k-blocking compute policies for the first three wavefront steps.

the start of the "mini-sweep". The parallel and SIMD efficiency achieved per node under this new k-block compute policy is the same as that seen when using a k_B value of N_z under the old policy, but permits processors to communicate prior to the completion of their entire $N_x/P_x \times N_y/P_y \times N_z$ volume. However, as shown in Figure 4(b), each processor must now receive two messages (as opposed to one) in order to have enough border data from upstream to complete the processing of a single data block.

5. Benchmark Comparisons for a Single Compute Node

5.1. Experimental Setup

The compiler configurations for all experiments performed in this section are given in Table 2. Although it is more usual to use the flag `arch=sm_20` when compiling for an NVIDIA GPU based on the Fermi architecture (such as the C2050), `arch=sm_13` provides better performance for our code. The value of k_B was set at N_z for all OpenCL implementations, to maximise SIMD efficiency, and at 1 for the native FORTRAN code (the default). All runs used the maximum number of cores available on the hardware.

5.2. Performance Impact of Memory and Work Distribution Strategies

Figure 5 shows the runtime of our OpenCL code for a Class C problem on one CPU (an Intel X5550), and on three GPUs (an NVIDIA Tesla C1060, an NVIDIA Tesla C2050 and an ATI FirePro V7800) for different values of the

Implementation	**Compiler**	**MPI**	**Flags**
FORTRAN 77	Sun Studio 12 (Update 1)	OpenMPI 1.4.2	`-O5 -native -xprefetch -xunroll=8 -xipo -xvector`
OpenCL	GCC 4.3	OpenMPI 1.4.2	`-O3 -funroll-loops`
CUDA (Host)	GCC 4.3	OpenMPI 1.4.2	`-O3 -funroll-loops`
CUDA (Device)	NVCC 4.0	N/A	`-O2 -arch="sm_13"`

Table 2: Compiler configurations for single-node experiments.

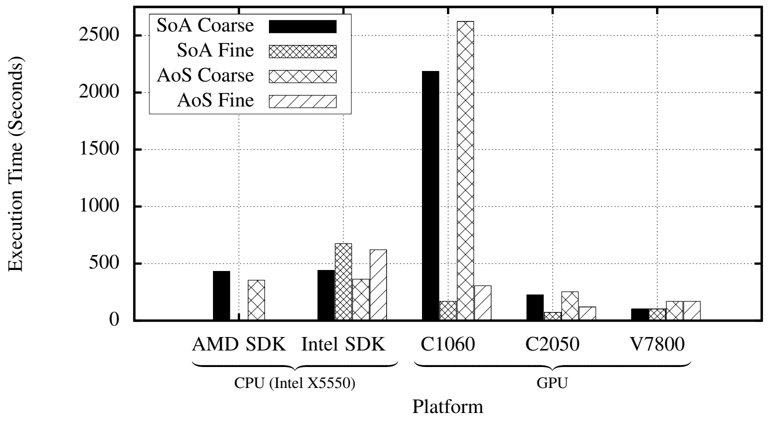


Figure 5: Performance comparison of different choices of memory layout and work-item distribution for Class C.

memory layout and work-item distribution parameters. We also compare the runtime on the CPU for the AMD and Intel SDKs.

These results highlight the importance of our chosen auto-tuning parameters. The coarse-grained work-item distribution is best suited to CPUs, achieving lower runtimes for both the AMD and Intel OpenCL SDKs, and the fine-grained work-item distribution is better suited to GPU architectures, giving the best runtime on all three GPU platforms. The architectures also favour different memory layouts as expected, and for the reasons explained in Section 4.3 – the CPU performs best when data is stored in AoS format, and the GPUs perform best when data is stored in SoA format. The Tesla C1060 is most affected by both parameters: the penalty for selecting the "wrong" work-item distribution is 12.9x; for memory layout it is 1.8x; and the difference between the best and worst performance on this architecture is 15.5x.

We do not present results for the AMD SDK using fine-grained work-item distribution on the CPU. On a Class A problem, this configuration was over 40x slower than the coarse-grained distribution – for a Class C problem, it exceeded an hour's wall time. The Intel SDK also takes a performance hit from the use of fine-grained distribution, but the impact is significantly lower. We do not know why this is the case, but conjecture it may be a result of pairing the AMD SDK with Intel hardware (which is not officially supported) or be related to a difference in the way the two SDKs map work-items to threads; further

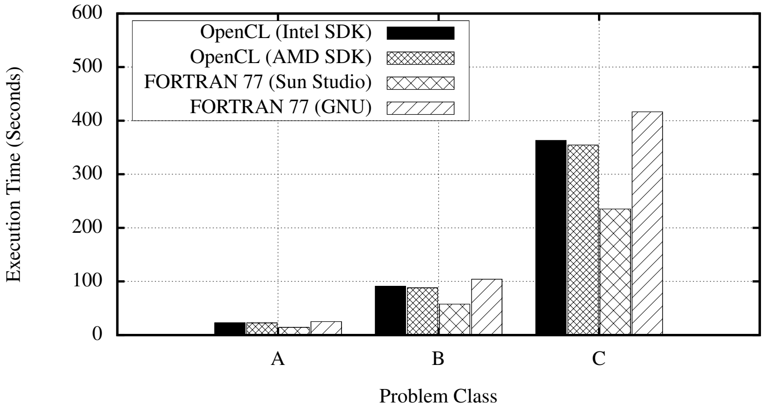


Figure 6: A comparison of OpenCL and FORTRAN 77 implementations of LU.

investigation into this matter is necessary.

5.3. Performance of OpenCL vs. Native Implementations

Next we consider the performance of the OpenCL implementation of LU against native FORTRAN 77 (for CPUs) and CUDA (for GPUs). It should be noted that this FORTRAN 77 implementation of LU is not the default implementation as provided by NASA, but one that was heavily optimised as part of our prior work [3] (this is also available at [22]). As stated, there are benefits in maintaining one single (OpenCL) source. However, this argument is more compelling if the performance of the OpenCL implementation is competitive with that of native implementations.

To ensure clarity of results, and unless otherwise stated, the OpenCL runtime configuration for each platform was selected as the best from the previous set of results (*i.e.* coarse-grained work-distribution with an AoS layout for CPUs, and fine-grained work-distribution with an SoA layout for GPUs).

The graph in Figure 6 compares the performance of the OpenCL and FORTRAN 77 implementations running on a quad-core Intel X5550 CPU. We present results from the same two OpenCL SDKs as previously, and from two FORTRAN 77 compilers; GCC, since it is the default compiler on most Linux systems and features regularly in performance studies of this nature, and Sun Studio, a heavy-weight industrial compiler capable of extensive code optimisations and thus representative of the compilers likely to be employed on production HPC clusters.

Following the trend reported elsewhere [5, 9], the OpenCL performance is marginally better (1.17x) than the FORTRAN 77 implementation compiled with GCC. When compiled with Sun Studio, the FORTRAN 77 code is 1.5x faster than OpenCL.

There is not likely to be one single reason for this performance gap, but rather several contributing factors. Firstly, the maturity of FORTRAN 77 compilers

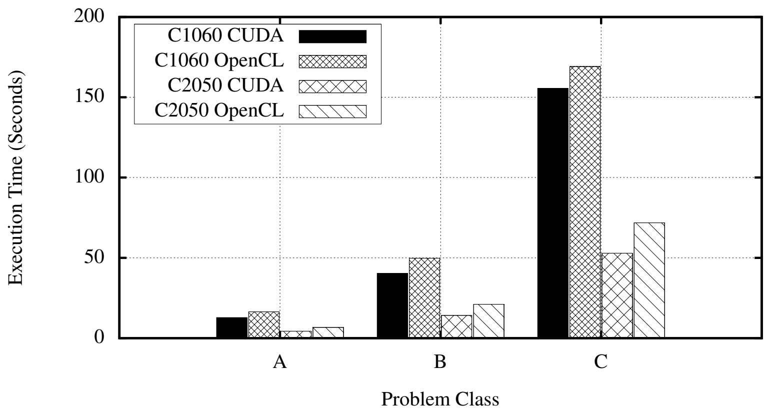


Figure 7: A comparison of OpenCL and CUDA implementations of LU.

is much greater than that of OpenCL compilers, having been tuned over many years for commodity-processor architectures. This is particularly true for x86, which has enjoyed binary compatibility over multiple processor generations. We attribute the difference between the GCC and Sun Studio compilers to the latter's ability to apply intensive inter-procedural optimisations (IPO) across all source files, increasing compile time significantly but often providing better runtime performance. Secondly, there are differences in memory behaviour: the OpenCL runtime creates threads which may operate on memory located in remote memory banks, whereas each of the MPI tasks in the FORTRAN 77 implementation operates only on its own, local, memory space. Finally, the implementation of the wavefront algorithm in the native code reflects its origin as a CPU-only code, in that the work of each MPI task is serial in nature (and not easily parallelisable). The implementation of the wavefront algorithm in the OpenCL code lends itself much better to parallelisation (and thus to portability across different architectures) but may suffer from increased synchronisation overhead (between kernels) and other inefficiencies.

The graph in Figure 7 compares the performance of the OpenCL and CUDA implementations of the benchmark. On the Tesla C1060, the CUDA implementation is marginally faster (1.08x); on the C2050 it is faster still (1.35x). Neither of these performance gaps are as large as those reported elsewhere for initial ports of CUDA codes [7, 8], but these have been shown to be due to differences between the optimisations carried out by NVIDIA's CUDA and OpenCL compilers – we believe that the small difference in performance shown here is a reflection of the hand-unrolled and hand-inlined nature of our implementation's most compute-intensive kernels.

The OpenCL runtime performance is clearly competitive with that of the FORTRAN 77 and CUDA implementations. OpenCL (and its associated compiler technology) are still relatively immature, and so these performance figures are likely to improve. Furthermore, we believe that the performance hit from

	Sierra	***Minerva*** (GPU Partition)
Nodes	1,944	6
CPUs/Node	2 × Intel X5660	2 × Intel X5650
Cores/Node	12	12
Core Frequency	2.8 GHz	2.66 GHz
Memory per Node	24 GB	24 GB
OS	CHAOS 4.4	SUSE Enterprise Linux 11
Interconnect	InfiniBand QDR (QLogic)	InfiniBand TrueScale 4X-QDR
Accelerators/Node	None	2 × NVIDIA M2050

Table 3: Hardware specifications of the *Sierra* and *Minerva* clusters.

Implementation	**Compiler**	**MPI**	**Flags**
FORTRAN 77	PGI 8.0.1	OpenMPI 1.3.2	`-O3 -Munroll -Minline`
OpenCL	PGI 8.0.1	OpenMPI 1.3.2	`-O3 -Munroll`
CUDA (Host)	GCC 4.5	OpenMPI 1.3.2	`-O3 -funroll-loops`
CUDA (Device)	NVCC 4.0	N/A	`-O2 -arch="sm_13"`

Table 4: Compiler configurations for multi-node experiments.

using OpenCL in the manner that we have described is small enough (on a single socket/node) that for many users it will be outweighed by the advantages of cross-platform code compatibility.

6. Benchmark Comparisons for Multiple Compute Nodes

6.1. Experimental Setup

We extend the study to consider multi-node performance. Two machines are used for these experiments: (*i*) the *Sierra* cluster housed in the Open Computing Facility at the Lawrence Livermore National Laboratory (LLNL), and (*ii*) the GPU partition of the *Minerva* cluster housed at the Centre for Scientific Computing at the University of Warwick. Hardware specifications for both machines can be found in Table 3, and compiler configurations are given in Table 4. Communication between nodes is conducted via MPI, with MPI ranks allocated in the following manner:

- **FORTRAN 77** – a single MPI rank is allocated per core;
- **OpenCL on CPUs** – a single MPI rank is allocated per node with OpenCL devices being used to execute computation within the node;
- **CUDA/OpenCL on GPUs** – a single MPI rank is allocated per GPU.

6.2. Performance Impact of k-blocking

Table 5 compares the performance of our OpenCL implementation with that of existing FORTRAN 77 and CUDA versions of the code. The FORTRAN 77 code uses a k_B of 1, and the CUDA code a k_B of $\min(N_x/P_x, N_y/P_y)$, as these values give the best performance; for OpenCL, we present results for all three of the k_B values discussed in Section 4.5. For the CPU, we give two results for

Sierra

Devices	OpenCL						FORTRAN 77
	$k_B = 1$		$k_B = \min\left(\frac{N_x}{P_x}, \frac{N_y}{P_y}\right)$		$k_B = N_z$		
12	1884.87	1246.45	235.36	242.45	235.36	242.45	107.78
24	1158.60	706.53	170.48	158.09	172.02	172.97	62.43
48	760.73	460.37	109.87	95.81	129.25	122.07	36.49
96	503.25	365.68	80.96	64.80	113.74	95.06	22.01
192	320.17	320.53	56.98	49.86	89.90	77.79	16.25

Minerva (GPU Partition)

Devices	OpenCL			CUDA
	$k_B = 1$	$k_B = \min\left(\frac{N_x}{P_x}, \frac{N_y}{P_y}\right)$	$k_B = N_z$	
1	1066.17	83.68	83.68	63.86
2	1514.91	102.08	96.62	72.51
4	1001.53	73.16	71.85	46.89
8	815.33	76.76	73.46	39.85

Table 5: Runtimes (in seconds) for multi-device executions of a Class C problem.

each k_B value: the left column contains results from the AMD SDK and the right column contains results from the Intel SDK.

On a single node, the native FORTRAN 77 code is an order of magnitude faster than the OpenCL implementation with $k_B = 1$, but this gap is closed to 2.2x for a more sensible choice of k_B. This performance difference is still larger than that seen on a single node in Section 5, and this is likely to be due to the fact that *Sierra* nodes are dual-socket. Since the OpenCL runtime groups multiple sockets in a system into a single device, we would expect the memory behaviour of a naïve OpenCL implementation to be worse in a multi-socket system (*e.g.* threads may access memory located at a different NUMA level, and cache coherence must be maintained across sockets). The difference between C2050 and M2050 hardware is minimal, and thus the native CUDA code remains 1.3x faster than OpenCL on a single GPU.

The performance gap is wider for both architectures at scale. The native code is 3.1x faster on 192 CPUs, and 1.8x faster on 8 GPUs. The reader's attention is also drawn to the poor scalability of LU, which results from the wavefront across processors – for the FORTRAN 77 implementation, the speed-up when strong-scaling from 12 to 192 cores is only 6.6x (out of an optimal 16x).

The results in this section demonstrate the importance of the k_B parameter, and show that it improves performance significantly across both architecture types. On the CPU, the best value of k_B gives up to an 8x speed-up, and on the GPU it gives a maximum speed-up of 12.8x; we would expect this performance improvement to be even larger for future architectures with more parallelism. One unexpected result is that a k_B value of N_z is actually fastest for the OpenCL code on the GPU, suggesting that the increased parallel efficiency afforded by the larger k-block is enough to outweigh the delay to downstream processors.

6.3. Performance Impact of the k-block Compute Policy

The effects of the k-block compute policy optimisation are quite varied. For $k_B = 1$, the performance of both the CPU and GPU implementations is improved considerably at low node counts. For all other k_B values, however, only the GPU continues to see any benefit (a modest improvement of around 5%) – all CPU implementations experience a slowdown (results not shown).

The new policy requires each processor to compute the values of some additional grid-points lying beyond the end of the current k-block (Figure 4(b)), and the relative cost of this additional compute is dependent upon the amount of parallelism supported by the hardware. In the worst case (where only one work-item can be processed at a time), the new policy will always result in a slowdown, since the hardware is unable to take advantage of the increase in exploitable parallelism; in the best case (where the number of work-items that can be processed is greater than or equal to $N_x/P_x \times N_y/P_y$) this additional compute will be carried out at little to no cost, since it will utilise hardware that is wasted when using the old policy.

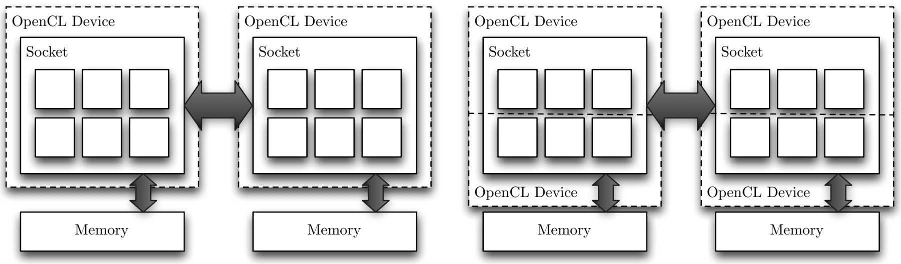
The amount of parallelism available in CPU hardware is likely to be less than the number of grid-points on the largest hyperplane, and thus the main impact of the new k-block compute policy is a significant increase in the amount of time the last processor $(P_x - 1, P_y - 1)$ spends waiting to receive its first message. GPUs execute many more threads with wider SIMD, and the compute time of each individual GPU is decreased, but they ultimately suffer from the same problem as CPUs at scale.

We intend to explore, through performance modelling and simulation, the effects of our new k-block compute policy in future work. We believe that there is much greater potential for improved performance with problem configurations not supported by LU, namely: problems where N_z is significantly larger than N_x or N_y, in which the cost of the first message reaching the last processor is amortized; and problems where computational cost far outweighs that of communication.

7. Device Fission

Although a multi-socket CPU node *can* be treated as a shared memory system, it is arguably unwise to do so in this case; a platform-agnostic implementation such as ours makes no consideration of the system's memory hierarchy and is therefore unlikely to exhibit good memory behaviour. More specifically, due to the fact that our OpenCL implementation has no control over the order in which work-groups and work-items are executed, nor the CPU cores to which they are allocated, it cannot guarantee that the set of work-groups processed by any given core will exploit either temporal or spatial cache locality.

An officially recognised extension to OpenCL known as *device fission* may go some way towards solving this problem, by allowing the runtime to "fission"



(a) Configuration 1 (b) Configuration 2

Figure 8: A dual-socket, hex-core node fissioned into (a) two sub-devices of six compute units; and (b) four sub-devices of three compute units.

a single device into multiple sub-devices. This extension – which is supported by the AMD and Intel SDKs, but not by the NVIDIA CUDA SDK – grants an OpenCL application the ability to assign work to specific cores.

In keeping with our policy of making only minimal changes to source, our implementation couples device fission with the existing MPI parallelism; instead of running one MPI task per node as before, we run one MPI task per created sub-device. This setup is demonstrated graphically in Figure 8 for the dual-socket, hex-core nodes present in *Sierra*. The advantage of this approach is that the only new code required is a simple check of whether fission is to be used in a given run, detection of the number of sub-devices to create, and the assignment of sub-devices to MPI tasks based on rank. As such, we argue that it is a logical way to add support for device fission into an existing MPI-based code.

The results in Table 6 show the effects of using two different device fission configurations in LU. We provide benchmarked execution times for all three of the k-block depths previously considered, and all experiments were run using the AMD SDK and the original k-block compute policy.

For a k-block value of 1, device fission provides significant performance improvements. The reader is reminded that without fission, the OpenCL code paired with the AMD SDK gave a runtime of 1158 seconds on 24 cores; the equivalent numbers for 2-way and 4-way fission are 568 and 394 seconds (speedups of 2x and 2.9x) respectively. It should also be noted that for matching core counts, the 4-way fission is always faster than the 2-way. This trend suggests that a 6-way or even 12-way device fissioning may give better performance still, with the domain decomposition across processors tending towards the default decomposition used by the FORTRAN code – unfortunately, due to the restriction that LU must be run on a number of devices that is a power of two, we were not able to investigate this further. For a k-block value of $\min(N_x/P_x, N_y/P_y)$, execution times are often lower with device fission than without, but the performance improvement is modest. For a k-block value of 162, execution times are higher, but this is to be expected; due to the increase in the number of OpenCL devices and MPI tasks, the time that each device spends waiting on

Configuration 1: 2-way Device Fission

Processor Cores	**MPI Ranks**	$k_B = 1$	$k_B = \min\left(\frac{N_x}{P_x}, \frac{N_y}{P_y}\right)$	$k_B = N_z$
24	4	568.56	161.13	180.23
48	8	397.41	106.72	144.61
96	16	276.68	83.06	118.13
384	64	123.75	34.76	82.85

Configuration 2: 4-way Device Fission

Processor Cores	**MPI Ranks**	$k_B = 1$	$k_B = \min\left(\frac{N_x}{P_x}, \frac{N_y}{P_y}\right)$	$k_B = N_z$
12	4	622.27	257.45	302.46
24	8	394.63	161.83	241.70
48	16	250.84	106.30	167.14
192	64	100.71	46.42	97.51

Table 6: Runtimes (in seconds) for two device fission configurations.

communication is higher.

We acknowledge that there are likely to be some overheads introduced by device fission, and that these may grow as the number of sub-devices created increases. Each MPI task will consume additional system resources (compared to a "pure" MPI implementation) to manage the task queue for its sub-device, and there may be other scheduling conflicts between threads created and managed by the OpenCL runtime. Specifically, it is unknown whether device fission guarantees that the affinity of a created sub-device will remain fixed, or whether it is possible for multiple MPI ranks to be assigned the same subset of cores. We intend to investigate alternative methods of using device fission, including n-way fissioning in nodes of n cores and using a single MPI task to manage a set of several sub-devices, in future work.

Nonetheless, the results in this section suggest that device fissioning can be a simple and effective way of improving the performance of hybrid MPI/OpenCL applications on densely packed CPU nodes – with minimal changes to source.

8. Discussion

8.1. Compiler Improvements

One of the key differences between OpenCL and native programming environments is that calls to OpenCL functions are made by way of a library. The fact that the operation of these functions is not necessarily defined until runtime restricts the optimisation freedom of the host compiler. Our results show that the IPO performed by the Sun Studio compiler has a considerable impact on performance, and this is exactly the sort of optimisation that will be difficult to employ for OpenCL code.

However, we do not see any reason that a hardware manufacturer supporting OpenCL could not produce a compiler that examines both the host code *and*

OpenCL code at compile time to produce a single, platform-specific binary. Although this would prevent said binary from being portable across different hardware, this is unlikely to be an issue for HPC – the fact that there is one source that is relatively easy to maintain may be much more important than a platform-agnostic binary. The makeup of a given supercomputer is fixed, and the hardware that is available in each node is likely to be known to a user at compile time. Such a compiler could potentially bring better performance to OpenCL codes.

8.2. Single Source Methodology

We acknowledge that the approach to maintaining a single-source application presented here is not guaranteed to work in every case. The performance gap between a legacy application and a naïve OpenCL implementation will be highly dependent upon the extent to which the original code was optimised, and re-designing an inherently serial application to target multiple parallel architectures will itself incur some cost. Further, we note that there will be particular areas of large codes for which it is necessary to maintain separate platform-specific source code for algorithmic reasons, or common functions (*e.g.* FFTs, matrix multiplication) for which highly optimised and platform-specific libraries are desirable.

However, we believe that the methodology demonstrated here is a simple way of achieving acceptable levels of performance portability across different architectures and that an approach such as ours will be well suited to many massively parallel workloads.

9. Conclusions

There is considerable interest in the use of accelerator-based architectures in high performance computing, not least because they promise high levels of performance at low cost. This paper supports this work by reporting on the first hybrid MPI/OpenCL implementation of LU, an application-level benchmark from the NPB Suite.

We report on the design of a single-source, platform agnostic code, and in particular on the performance impact of memory layout and work-item/work-group distribution strategies. We demonstrate that a poorly chosen value for either of these parameters can lead to up to a 12.9x decrease in performance, and selecting the worst value in both cases can cause a slowdown of as much as 15.5x. The use of k-blocking is shown to improve performance of wavefront codes utilising a SIMD programming model by up to 8x on CPUs and 12.8x on GPUs; alternatively, the use of 4-way device fission on CPUs gives a speed-up of 3x.

Our OpenCL implementation is competitive with native FORTRAN 77 and CUDA implementations running on the same hardware, performing 1.3–1.5x slower on a single node. This performance gap widens at scale due to differences in communication behaviour and the wavefront algorithm employed across MPI ranks. Whether or not this performance compromise is acceptable is open to debate, but will ultimately be decided by HPC sites; it is not simply a question of absolute performance, but one of time, money and protecting investment.

The platform agnostic-approach to HPC code development described here recognises key differences between architectures, and dictates that these differences are accounted for during development. However, the approach allows for such changes to be made incrementally, and to a single source, without the need to maintain completely separate code-paths for each platform. It therefore represents a productive and forward-thinking approach to HPC code development, offering a middle-ground between (*i*) focusing development on a single architecture today, hoping that the resulting code will execute effectively on future hardware; and (*ii*) potentially wasting considerable development effort on writing efficient code for each new hardware offering.

Acknowledgements

We are grateful to Scott Futral, Jan Nunes, Todd Gamblin and the Livermore Computing Team for access to, and help in using, the *Sierra* machine located at the Lawrence Livermore National Laboratory; and would also like to thank the University of Warwick Centre for Scientific Computing for access to *Minerva*.

Funding

This work is sponsored in part by The Royal Society through their Industry Fellowship Scheme (IF090020/AM).

Access to the LLNL Open Computing Facility is made possible through collaboration with the UK Atomic Weapons Establishment under grants CDK0660 (The Production of Predictive Models for Future Computing Requirements) and CDK0724 (AWE Technical Outreach Programme). The HPC performance benchmarking research is also supported jointly by AWE and the TSB Knowledge Transfer Partnership grant number KTP006740.

Sandia National Laboratories is a multiprogram laboratory managed and operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

[1] The OpenCL Specification, `http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf` (June 2011).

[2] D. Bailey, et al., The NAS Parallel Benchmarks, Tech. Rep. RNR-94-007, Moffet Field, CA (1994).

[3] S. J. Pennycook, S. D. Hammond, G. R. Mudalige, S. A. Jarvis, Performance Analysis of a Hybrid MPI/CUDA Implementation of the NAS-LU Benchmark, SIGMETRICS Performance Evaluation Review 38 (4) (2011) 23–29.

[4] S. J. Pennycook, S. D. Hammond, G. R. Mudalige, S. A. Wright, S. A. Jarvis, On the Acceleration of Wavefront Applications using Distributed Many-Core Architectures, The Computer Journal 55 (2) (2012) 138–153.

[5] S. McIntosh-Smith, T. Wilson, J. Crisp, A. Ávila Ibarra, R. B. Sessions, Energy-Aware Metrics for Benchmarking Heterogeneous Systems, SIGMETRICS Performance Evaluation Review 38 (2011) 88–94.

[6] S. Wienke, D. Plotnikov, D. an Mey, C. Bischof, A. Hardjosuwito, C. Gorgels, C. Brecher, Simulation of Bevel Gear Cutting with GPGPUs – Performance and Productivity, Computer Science - Research and Development (2011) 1–10.

[7] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, J. Dongarra, From CUDA to OpenCL: Towards a Performance-Portable Solution for Multiplatform GPU Programming, Tech. Rep. UT-CS-10-656, Knoxville, TN (2010).

[8] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, H. Kobayashi, Evaluating Performance and Portability of OpenCL Programs, in: Proceedings of the Fifth International Workshop on Automatic Performance Tuning, 2010.

[9] R. Weber, A. Gothandaraman, R. J. Hinde, G. D. Peterson, Comparing Hardware Accelerators in Scientific Applications: A Case Study, IEEE Transactions on Parallel and Distributed Systems 22 (1) (2011) 58–68.

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

[11] G. R. Mudalige, M. K. Vernon, 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, Miami, FL, 2008.

[12] S. Saini, D. H. Bailey, NAS Parallel Benchmark (Version 1.0) Results 11-96, Tech. Rep. NAS-96-18, Moffet Field, CA (1996).

[13] A. M. Aji, W. C. Feng, Accelerating Data-Serial Applications on GPGPUs: A Systems Approach, Tech. Rep. TR-08-24, Blacksburg, VA (2008).

[14] C. Gong, J. Liu, Z. Gong, J. Qin, J. Xie, Optimizing Sweep3D for Graphic Processor Unit, in: Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing, Busan, Korea, 2010, pp. 416–426.

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

[16] Y. Munekawa, F. Ino, K. Hagihara, Design and Implementation of the Smith-Waterman Algorithm on the CUDA-Compatible GPU, in: Proceedings of the IEEE International Conference on Bioinformatics and Bioengineering, Athens, Greece, 2008, pp. 1–6.

[17] F. Petrini, G. Fossum, J. Fernandez, A. L. Varbanescu, N. Kistler, 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, Long Beach, CA, 2007.

[18] AMD Accelerated Parallel Processing SDK, `http://developer.amd.com/sdks/AMDAPPSDK/` (June 2011).

[19] Intel OpenCL SDK, `http://software.intel.com/en-us/articles/opencl-sdk/` (June 2011).

[20] CUDA Toolkit 4.0, `http://developer.nvidia.com/cuda-toolkit-40` (May 2011).

[21] M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, P. H. J. Kelly, Performance Analysis of the OP2 Framework on Many-core Architectures, SIGMETRICS Performance Evaluation Review 38 (2011) 9–15.

[22] PCAV High Performance Computing Downloads, `http://www2.warwick.ac.uk/fac/sci/dcs/research/pcav/areas/hpc/download` (September 2011).

John Pennycook is a Ph.D. student in the Performance Computing and Visualisation Group at the University of Warwick. His research interests are: the porting and optimisation of high-performance scientific codes for novel architectures, such as graphics processors and the Intel Many-Integrated Core architecture; investigating the performance impact of alternative programming methodologies; and building analytical performance models for real-world scientific applications and benchmarks. He graduated from the University of Warwick in 2009 with a B.Sc. (Hons) in Computer Science.

Simon Hammond is a member of the Scalable Computer Architectures department at Sandia National Laboratories. His research focuses on the analysis, modelling and porting of large-scale parallel applications to future computing architectures as well as performance oriented middleware and code optimisation. Prior to joining Sandia, Simon was a joint Research Fellow at the University of Warwick and AWE plc where he worked on application simulation and code ports for capability computing environments. He received a Ph.D. from the University of Warwick for this work in 2011.

Steven Wright is a Ph.D. student in the Performance Computing and Visualisation group at the University of Warwick. Before undertaking his Ph.D., Steven recieved a First Class Master of Engineering degree in Computer Science, also from the University of Warwick. Steven is interested in all aspects of High Performance Computing, but the primary focus of his Ph.D. is in Parallel I/O acceleration, simulation and optimisation. He is also interested in the development of optimised I/O libraries and Parallel File Systems.

Andy Herdman is a Senior Research Leader in the High Performance Computing section at AWE plc. He graduated from Newcastle University with a B.Sc. (Hons) in Mathematics and Statistics; he also holds an M.Sc. in Applied Mathematics. He has worked in the HPC applications field for over 15 years. Roles have included Technical Team Leader, responsible for code porting and optimisation, and Group Leader of multi-disciplined teams, covering engineering codes, data visualisation, and advanced technologies. He is currently a Visiting Fellow at the University of Warwick, researching in the field of performance engineering.

Iain Miller is a Computer Scientist at AWE plc, with primary interests in the benchmarking and modelling of high performance computer systems. He is also working towards a Ph.D. in Applied Computational Intelligence with the University of the West of Scotland.

Stephen Jarvis is Professor of High Performance Computing at the University of Warwick and co-organiser for one of the UK's High End Scientific Computing Training Centres. He has authored more than 130 refereed publications (including three books) and has been a member of more than 50 programme committees for IEEE/ACM international conferences and workshops since 2003, including: IPDPS, HPDC, CCGrid, SC, DSN, ICPP. He is a former member of the University of Oxford Computing Laboratory, and in 2009 was awarded a prestigious Royal Society Industry Fellowship in support of his industry-focused work on HPC.