Towards Portable Performance for Explicit Hydrodynamics Codes

A. C. Mallinson
Performance Computing and Visualisation
Department of Computer Science
University of Warwick, UK
acm@dcs.warwick.ac.uk

D. A. Beckingsale
Performance Computing and Visualisation
Department of Computer Science
University of Warwick, UK
dab@dcs.warwick.ac.uk

W. P. Gaudin
High Performance Computing
UK Atomic Weapons Establishment
Aldermaston, UK
Wayne.Gaudin@awe.co.uk

J. A. Herdman
High Performance Computing
UK Atomic Weapons Establishment
Aldermaston, UK
Andy.Herdman@awe.co.uk

S. A. Jarvis
Performance Computing and Visualisation
Department of Computer Science
University of Warwick, UK

ABSTRACT

Significantly increasing intra-node parallelism is widely recognised as being a key prerequisite for reaching exascale levels of computational performance. In future exascale systems it is likely that this performance improvement will be realised by increasing the parallelism available in traditional CPU devices and using massively-parallel hardware accelerators. The MPI programming model is starting to reach its scalability limit and is unable to take advantage of hardware accelerators; consequently, HPC centres (such as AWE) will have to decide how to develop their existing applications to best take advantage of future HPC system architectures. This work seeks to evaluate OpenCL as a candidate technology for implementing an alternative hybrid programming model, and whether it is able to deliver improved code portability whilst also maintaining or improving performance. On certain platforms the performance of our OpenCL implementation is within 4% of an optimised native version.

Categories and Subject Descriptors

D.1.3 [**Software**]: Concurrent Programming—*Parallel Programming*

General Terms

Performance, Experimentation, Languages

Keywords

Hydrodynamics, OpenCL, MPI, Portability, High Performance Computing

1. INTRODUCTION

Significantly increased intra-node parallelism, relative to current HPC architectures, is recognised as a feature of future exascale systems [11]. This increase in parallelism is likely to come from two sources: (i) increased CPU core counts; and (ii) massively-parallel hardware accelerators, such as discrete general-purpose graphics processing units (GPGPUs), field-programmable gate arrays (FPGAs), accelerated processing units (APUs) or co-processors (e.g. Intel Xeon Phi). The OpenCL specification provides a programming framework that supports all of these architectures, and guarantees functionality portability of valid OpenCL programs. There is, however, no guarantee of performance portability [1].

Large HPC centres have made a significant investment in maintaining their existing scientific codebases, which typically use the message passing interface (MPI) to provide parallelism at both the inter- and intra-node levels. However, using MPI alone can create significant problems at scale [7]. As the number of processors rises, the amount of memory that the MPI runtime requires becomes prohibitive. This problem is likely to get worse as the amount of memory available to each processor core decreases, in keeping with current trends in HPC system design. Furthermore, MPI provides no mechanism for codes to use attached accelerator devices, which will be critical for achieving high levels of performance on future systems.

Hybrid programming models provide a way to extend MPI-based applications and provide a means through which new hardware accelerator devices can be utilised. In this context, a hybrid programming model consists of MPI for inter-node communication, together with one or more complementary technologies for intra-node parallelism. Due to the large, complex codebases of many scientific applications, incrementally incorporating a hybrid approach is more feasible than rewriting an application from scratch. Potential intra-node technologies appropriate for this model include OpenMP, OpenACC, OpenCL and CUDA.

This work seeks to evaluate OpenCL as a candidate technology for implementing the hybrid programming model. In particular, we examine its utility for delivering portable performance across a range of architectures. We integrate OpenCL with an existing MPI based Fortran code, CloverLeaf, and analyse the performance of the code across a range of modern HPC architectures. Specifically, we make the following key contributions:

- We present a detailed description of CloverLeaf's hydrodynamics algorithm, and its implementation in OpenCL;
- We present a performance comparison of the OpenCL version of the code, against optimised native versions, on a range of modern architectures, including CPUs from Intel and AMD, GPUs from Nvidia and AMD, and an APU from AMD;
- Finally, we present a number of optimisations to improve both the performance and portability of our OpenCL implementation of CloverLeaf.

The remainder of this paper is organised as follows: Section 2 discusses related work in this field; Section 3 provides some background information on the hydrodynamics scheme employed by the CloverLeaf mini-application and the OpenCL programming model; Section 4 describes our OpenCL implementation of CloverLeaf together with information on the optimisations we have examined; the results of our study and our experimental setup are presented in Section 5; finally, Section 6 concludes the paper and outlines plans for future work.

2. RELATED WORK

To date, insufficient work has been undertaken to examine whether OpenCL is a viable alternative programming model for delivering intra-node parallelism on HPC system architectures. This is the case for scientific applications in general, as well as for the Lagrangian-Eulerian explicit hydrodynamics applications that are the focus of our work. Even less work exists which examines whether OpenCL can effectively deliver portable performance for these hydrocodes across a range of current architectures. This is a key aim of our study.

In previous work we examined alternative approaches (CUDA, OpenCL, and OpenACC) for porting a Lagrangian-Eulerian explicit hydrodynamics application to GPU-based systems [16]. A considerable body of work also exists which has examined porting Smoothed Particle Hydrodynamics (SPH) applications to GPU-based systems [14, 17, 24, 25]. These SPH applications employ mesh-less, particle based, Lagrangian numerical methods and are therefore significantly different to the hydrodynamics scheme used by our code. Studies involving SPH have also predominantly focused on utilising CUDA and have not sought to examine OpenCL as an alternative technology for delivering portable performance. Whilst Bergen *et al.* produce an OpenCL version of a finite-volume hydrodynamics application which is similar to that involved in our work, they do not present any performance results or compare the development, performance or portability of the application to alternative approaches or across architectures [8]. The GAMER library also provides similar functionality to that employed here, however it is implemented entirely in CUDA and therefore does not allow the evaluation of OpenCL as an alternative approach for delivering portable performance [26].

Brook *et al.* present their experiences of porting two computational fluid dynamics (CFD) applications to an accelerator (a Euler-based solver and a BGK model Boltzmann solver) [9]. Whilst the Euler-based solver is similar to the application documented in our work, they focus on the Intel Xeon Phi architecture and employ only the OpenMP programming model.

Existing work has also examined using OpenCL to deliver portable performance within other scientific domains. Pennycook presents details of the development of an OpenCL implementation of the NAS LU benchmark [21] and a molecular dynamics application [22] which achieve portable performance across a range of current architectures. Similarly, Brown *et al.* describe work and performance results within the molecular dynamics domain which enables computational work to be dynamically distributed across both CPU and GPU architectures. Both OpenCL and CUDA are compared in their study [10].

Du *et al.* [12] and Weber *et al.* [29] provide direct analyses of OpenCL's ability to deliver portable performance for applications targeting accelerator devices. Both however, focus on different scientific domains to our work; linear algebra routines and Quantum Monte Carlo simulations, respectively. Additionally, Komatsu *et al.* [20] and Fang *et al.* [13] provide detailed examinations of the performance differences between CUDA and OpenCL, as well as OpenCL's ability to deliver portable performance. In his master's thesis, van der Sanden evaluates the performance portability of several image processing applications expressed in OpenCL and provides several techniques for achieving portable performance [28].

Whilst the vast majority of existing work focuses on OpenCL applications on accelerator devices, Karrenberg and Hack also present several techniques for improving the performance of OpenCL codes on current CPU devices [18].

Existing studies have also examined utilising OpenCL together with message passing technologies such as MPI to deliver portable performance across a cluster [23, 27]. However these studies focused on applications from different scientific domains to our work; the Finite-Difference Time-Domain method and molecular dynamics respectively. Kim *et al.* propose a novel framework (SnuCL) which also enables OpenCL applications to run in a distributed manner across a cluster [19].

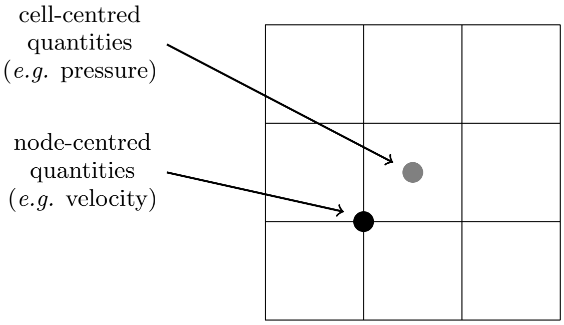

Figure 1: The *staggered grid* used by CloverLeaf to solve Euler's equations.

3. BACKGROUND

In this section we provide details on the Hydrodynamics scheme employed by CloverLeaf, and an overview of the OpenCL programming model.

3.1 Hydrodynamics Scheme

CloverLeaf uses a Lagrangian-Eulerian scheme to solve Euler's equations of compressible fluid dynamics in two spatial dimensions. The equations are solved on a *staggered grid* (see Figure 1) in which each cell centre stores the quantities energy, density, and pressure, and each node stores a velocity vector. An explicit finite-volume method is used to solve the equations with second-order accuracy.

The solution is advanced forward in time repeatedly until the desired end time is reached. One iteration, or timestep, of CloverLeaf proceeds as follows: (i) a Lagrangian step advances the solution in time using a predictor-corrector scheme, with the cells becoming distorted as the nodes move due to the fluid flow; (ii) an advection step restores the cells to their original positions by moving the nodes, and calculating the amount of material that has passed through each cell.

The two main steps of the application can be divided into a number of individual kernels. In this instance, we use *kernel* to refer to a self-contained function which carries out one specific step of the hydrodynamics algorithm. Each kernel iterates over the staggered grid, updating the appropriate quantities using the required stencil. Figure 2 shows the Fortran code for one of the fourteen kernels that carry out the hydrodynamic timestep. The kernels contain no subroutine calls and avoid using complex features like Fortran's derived types, making them ideal candidates for porting to OpenCL.

Not all of the kernels used by CloverLeaf are as simple as the example in Figure 2. However, during the initial development of the code, we re-engineered the algorithm to ensure that all loop-level dependencies in the kernels were eliminated. Most of the dependencies were removed via small code rewrites: large loops were broken into smaller parts, extra temporary storage was employed where necessary, and branches inside loops were also removed. Typically, branches within the original loop blocks were used for error-detection, however, since CloverLeaf is a mini-application that handles a fixed set of robust problems, these checks can be removed without affecting the hydrodynamics scheme.

```
DO k = y_min, y_max
  DO j = x_min, x_max

    p(j,k) = (1.4-1.0)*d(j,k)*e(j,k)
    pe = (1.4-1.0)*d(j,k)
    pv = -d(j,k)*p(j,k)

    v = 1.0/d(j,k)
    ss2 = v*v*(p(j,k)*pe-pv)

    ss(j,k)=SQRT(ss2)

  END DO
END DO
```

Figure 2: One of CloverLeaf's fourteen hydrodynamics kernels, `ideal_gas`. Note the simple 1-point stencil used.

Each CloverLeaf kernel can have multiple implementations. For example, both C and Fortran kernels have previously been developed for the entire hydrodynamics scheme. The design of CloverLeaf enables the desired kernel to be selected at runtime. Calling these kernels is managed by a driver routine (see Figure 3), which also handles data communication and I/O. In order to produce the OpenCL version of CloverLeaf, we developed a new implementation for each of these kernel functions.

3.2 OpenCL

OpenCL is an open standard that enables parallel programming of heterogeneous architectures. Managed by the Khronos group and implemented by over ten vendors—including AMD [2], Intel [3], IBM [4], and Nvidia [5]—OpenCL code can be run on many architectures without recompilation. Each compiler and runtime is, however, at a different stage of maturity, so performance currently varies between vendors.

The programming model used by OpenCL is similar to Nvidia's CUDA model. Therefore, mapping OpenCL programs to GPU architectures is straightforward. The best way to map OpenCL programs to CPU architectures, however, is less clear.

The OpenCL programming model distinguishes between a *host* CPU and an attached accelerator *device* such as a GPU. The host CPU runs code written in C or C++ that makes

```
IF (use_fortran_kernels) THEN

    CALL ideal_gas_kernel

ELSEIF (use_c_kernels) THEN

    CALL ideal_gas_kernel_c

ELSEIF (use_opencl_kernel) THEN

    CALL ideal_gas_kernel_opencl


                   ⋮
ENDIF
```

Figure 3: Runtime kernel selection for the `ideal_gas` kernel.

```
try {
    ideal_knl.setArg(0, x_min);
                    ⋮
    if (predict == 0) {
        ideal_knl.setArg(4,
                CloverCL::density1_buffer);
    } else {
        ideal_knl.setArg(4,
                CloverCL::density0_buffer);
    }
} catch(cl::Error err) {
    CloverCL::reportError(err, ...);
}

CloverCL::enqueueKernel(ideal_knl, x_min, x_max,
                        y_min,y_max);
```

(a) The OpenCL C++ host side code for the `ideal_gas` kernel.

```
for (int k = get_global_id(1); k <= y_max;
        k += get_global_size(1)) {
    for (int j = get_global_id(0); j <= x_max;
            j += get_global_size(0)) {
        double ss2,v,pe,pv;

        p[ARRAY2D(j,k,...)]= (1.4-1.0)
            *d[ARRAY2D(j,k,...)]
            *e[ARRAY2D(j,k,...)];

        pe=(1.4-1.0)*d[ARRAY2D(j,k,...)];
        pv=-d[ARRAY2D(j,k,...)] *p[ARRAY2D(j,k,...)];

        v = 1.0/d[ARRAY2D(j,k,...)];
        ss2=v*v*(p[ARRAY2D(j,k,...)]*pe-pv);

        ss[ARRAY2D(j,k,...)]=sqrt(ss2);
    }
}
```

(b) The OpenCL device code for the `ideal_gas` kernel.

Figure 4: The two components of the OpenCL version of the `ideal_gas` kernel. Branching is performed on the host, and the device code closely mirrors the original Fortran.

function calls to the OpenCL library in order to control, communicate with, and initiate tasks on one or more attached devices, or on the CPU itself. The target device or CPU runs functions (*kernels*) written in a subset of C99, which can be compiled just-in-time, or loaded from a cached binary if one exists for the target platform. OpenCL uses the concepts of *devices*, *compute units*, *processing elements*, *work-groups*, and *work-items* to control how OpenCL kernels will be executed by hardware. The mapping of these concepts to hardware is controlled by the OpenCL runtime.

Generally, an OpenCL device will be an entire CPU socket or an attached accelerator. On a CPU architecture, both the compute units and processing elements will be mapped to the individual CPU cores. On a GPU this division can vary, but compute units will typically map to a core on the device, and processing elements will be mapped to the functional units of the cores.

```
int k = get_global_id(1);
int j = get_global_id(0);

if ( (j<=x_max) && (k<=y_max) ) {
        double ss2,v,pe,pv;

        p[ARRAY2D(j,k,...)]= (1.4-1.0)
            *d[ARRAY2D(j,k,...)]
            *e[ARRAY2D(j,k,...)];

        pe=(1.4-1.0)*d[ARRAY2D(j,k,...)];
        pv=-d[ARRAY2D(j,k,...)] *p[ARRAY2D(j,k,...)];

        v = 1.0/d[ARRAY2D(j,k,...)];
        ss2=v*v*(p[ARRAY2D(j,k,...)]*pe-pv);

        ss[ARRAY2D(j,k,...)]=sqrt(ss2);
}
```

Figure 5: The new device code for the `ideal_gas` kernel.

Each kernel is executed in a Single Program Multiple Data (SPMD) manner across a one, two or three dimensional range of work-items, with collections of these work-items being grouped together into work-groups. Work-groups map onto a compute unit and the work-items that they contain are executed by the compute unit's associated processing elements. The work-groups which make up a particular kernel can be dispatched for execution on all available compute units in any order. On a CPU, the processing elements of the work-group will be scheduled across the cores using a loop. If vector code has been generated, the processing elements will be scheduled in SIMD, using the vector unit of each CPU core. On a GPU, the processing-elements run work-items in collections across the cores, where the collection size or width depends on the device vendor; Nvidia devices run work-items in collections of 32 whereas AMD devices use collections of 64 work-items.

OpenCL is therefore able to easily express both task and data parallelism within applications. The OpenCL programming model provides no global synchronisation mechanism between work-groups, although it is possible to synchronise within a work-group. This enables OpenCL applications to scale up or down to fit different hardware configurations.

4. IMPLEMENTATION

Integrating OpenCL with Fortran is not trivial as the C and C++ bindings described by the OpenCL standard are not easy to call directly from Fortran. In order to create the OpenCL implementation of CloverLeaf, we wrote a new OpenCL-specific version for each of the existing kernel functions.

The implementation of each kernel is split into two parts: (i) an OpenCL device-side kernel that performs the required mathematical operations and; (ii) a host-side C++ routine to set up the actual OpenCL kernel. The Fortran driver routine calls the C++ routine, which is responsible for transferring the required data, setting kernel arguments, and adding the device-side kernel to the OpenCL work-queue with the appropriate work-group size.

Since each kernel performs a well defined mathematical function, and the Fortran versions avoid the use of any complex language features, writing the OpenCL kernels is almost a direct translation. To finalise the OpenCL kernels however, a few changes needed to be made to produce the initial implementation (see Figure 4b). The loops over the staggered grid (see Figure 2) were re-factored to account for the fact that one work-item is launched per grid point. Consequently the lower-bound of each loop became the `global ID` of the individual work-items and the upper-bound of each loop became the `global size` of the particular work-group. To ensure that each cell is only updated once, the loop increments by the global size every iteration. This loop allows the kernel to produce the correct answer when launched with less work items than there are cells in the problem. In order to produce comparable results to the Fortran kernel, all computation is performed in double precision.

4.1 Implementation Decisions & Initial Optimisations

Each C++ setup routine relies on a static class, `CloverCL`, which provides common functionality for all the different setup routines. We moved as much logic as possible from the actual kernel functions into this static class. This helped to ensure that particular pieces of logic (e.g. the kernel `setArg` commands) are only re-executed when absolutely necessary thus improving overall performance.

The OpenCL buffers and kernels are created, stored and managed from within this class, allowing buffers to be shared between kernels. This buffer sharing was particular important for obtaining high performance across all architectures. It also facilitated achieving full device residency of the OpenCL implementation on architectures which are constructed from accelerator based devices (e.g. GPGPUs) attached via a PCI bus to the main system nodes. Achieving full device residency and thus minimising data movement across the relatively slow PCI bus was crucial in achieving high performance on many current architectures.

The use of OpenCL `wait` operations was also minimised in the initial implementation via the use of a single in-order work-queue and global event objects stored in the static class (`CloverCL`). This enabled a dependency chain to be established between the invocations of each kernel within each timestep of the algorithm. The overall algorithm thus proceeds by continually adding kernel invocations to the work-queue in the order that they are required to be executed, with the queue's in-order properties providing the necessary synchronisation between the various kernels.

The static class also contains other methods that provide an additional layer of abstraction around common OpenCL routines. One method, `enqueueKernel`, provides a wrapper around `enqueueNDRangeKernel`. By passing all calls that add a kernel to the work-queue through this function, we can ensure that the number of work items launched will always be a multiple of the preferred work-group multiple[1]. We found this approach delivered important performance benefits with the initial implementation by improving the execution efficiency of the OpenCL kernels on several current hardware devices.

All Fortran intrinsic operations (such as `SIGN, MAX etc.`) were also replaced with the corresponding OpenCL built-in function to ensure optimal performance.

The majority of the control code in the original Fortran kernels was moved into the C++ setup routines. Figure 4a illustrates this for the `ideal_gas` kernel. This ensures that branching is also always performed on the host instead of on any attached device, enabling the device kernels to avoid stalls and thus maintain higher levels of performance.

	Cray XK6
Processor	AMD Opteron 6272
GPU	Nvidia X2090
Compute Nodes	40
CPUs/Node	1
GPUs/Node	1
Total CPUs	40
Total GPUs	40
CPU Memory/Node	32GB
GPU Memory/Node	6GB
Interconnect	Cray “Gemini”
Compilers	Cray 4.1.40, GNU 4.7.2
MPI	Cray MPI (Mpich2) v5.6.2.2
OpenCL/CUDA	Nvidia CUDA Toolkit 5.0

Table 2: Summary of Cray XK6 hardware.

To improve the performance of the OpenCL/MPI integration in CloverLeaf we employed the OpenCL built-in `clEnqueueReadBufferRect` function to read back only the minimum amount of required data from the OpenCL buffers, directly into the MPI communication buffers. The original data ordering within the MPI communication buffers was altered to better integrate with the `clEnqueueReadBufferRect` function. This removes the requirement to explicitly pack the communication buffers on the target device using a separate OpenCL kernel and makes use of an optimised OpenCL built-in function. Similarly we also employ the OpenCL `clEnqueueWriteBufferRect` function for transferring data back to the OpenCL buffers following an MPI communication operation.

4.2 Reduction Operations

Reduction operations are required by CloverLeaf in two locations: timestep control, and summary printouts of intermediate results. Since the timestep value is calculated frequently, it is crucial we utilise a high performance reduction implementation. We were unable to find a general optimised reduction operation written in OpenCL, and consequently had to implement our own reduction operations.

The OpenCL reduction functions were implemented as separate kernels. Their operation differs significantly from the Fortran and C kernels which either use nested loops to iterate over the entire source array, or OpenMP reduction primitives. Due to the architectural differences between CPUs and GPUs, we developed two OpenCL reduction kernels. Whilst these kernels are not portable across architectures it makes sense to specialise them, as reductions are fundamental to scientific applications and the kernels could be reused across many applications. Ultimately, our view is that reduction operations should be provided by a library, and therefore specialising them would not affect the portability of the actual application code.

The reduction kernel that targets GPU devices is based on work presented by Harris, although we generalise his method to arbitrary sized arrays [15]. We employ a multi-level tree-based approach in which kernel launches are used as synchronisation points between levels of the tree. The tree con-

[1] `CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE`

	Discrete CPUs		Integrated CPU/GPU	Discrete GPUs	
	Xeon E3-1240	Opteron 6272	Trinity A10-5800K	Tesla K20	FirePro V7800
Manufacturer	Intel	AMD	AMD	Nvidia	AMD
Compute Units	8	16	6	13	18
Proc. Elements	8	16	384	2496	1440
Peak SP[a] (GFlop/s)	211	-	-	3520	2000
Peak DP[b] (GFlop/s)	-	147	-	1170	400
Clock Speed (GHz)	3.3	2.1	0.8	0.705	-
Mem. Capacity (GB)	16	32	2	5	2
Mem. Speed (GHz)	1.333	1.6	1.866	-	-
Mem. Bandwidth (GB/s)	21	36.5	-	208	128
Total Drawn Power (W)	80	115	100	-	138
Host Compilers	Intel 12.1 & 13.1	Cray 4.1.40[c]	GNU 4.7.2	GNU 4.7.2	Intel 13.1
Intel Host Flags	`-O3 -ipo -no-prec-div -restrict -fno-alias -fp-model strict -fp-model source -prec-div -prec-sqrt`				
GNU Host Flags	`-O3 -march=native -funroll-loops`				
Cray Host Flags	`-em -ra -h`				
OpenMP Libraries	Cray, Intel				
Intel OpenCL SDK	Intel OpenCL 2012 & 2013				
AMD OpenCL SDK	AMD APP 2.7				
Nvidia SDK	CUDA Toolkit 5.0				
OpenCL Flags	`-cl-mad-enable -cl-fast-relaxed-math`				
CUDA Flags	`-gencode arch=compute_30, code=sm_35`				

[a] Single Precision
[b] Double Precision
[c] GNU 4.7.2 used for OpenCL runs.

Table 1: Hardware and software configuration.

tinues until the input to a particular level is small enough to fit within one OpenCL work-group on a given device. In the final level of the tree a single work-group is launched on one compute unit of the associated device, which then calculates the final result of the reduction operation. In all stages each work-item within the reduction kernel reads two values from global memory and applies the binary reduction operator to them, storing the result in local memory. These global memory operations are aligned on the device's preferred vector width to enable the coalescing of memory operations and to ensure efficient bandwidth utilisation.

Next, a tree-based reduction occurs on the partial results stored within the local memories. In this phase the number of active threads is halved in each iteration, until all the partial results have been reduced to one single value. Here the local memory references are arranged to avoid memory bank conflicts to ensure efficient bandwidth utilisation. This single value is then written by one thread back to global memory for the next level of the reduction tree to operate on.

To reduce the number of levels within the reduction tree (and thus the number of kernel launches) we maximise the number of work-items launched within each particular work-group. Thus, for each work-group, the number of input values read from global memory into the local memories is maximised, relative to the one value written back to global memory. We always ensure that the number of work-items launched for the reduction kernels is a power of 2, and an exact multiple of the preferred vector width of the device. Our implementation generalises to handle arbitrary sized arrays by limiting, if required, the number of data values read from global memory by the last work-group. Instead, work-items beyond this limit insert dummy values into their corresponding local memory locations. This ensures that the tree-based part of the reduction is always balanced.

The reduction kernel that targets CPU devices operates in a similar manner. Here, we employ a two-level hierarchical approach in which kernel launches are again used to synchronise between the two levels. In the first level, the input array is partitioned so that it is distributed as evenly as possible across all the available cores of the CPU. If required, the last work-group is again limited to handle uneven distributions of arbitrary sized arrays. Only one work-item is launched for each core of the associated CPU and all work-groups are initialised to only contain one work-item. Each work-item then sequentially reduces the data values within the portion of the input array assigned to it and stores the resultant value back into memory. The number of partial results output from this phase is therefore equal to the number of cores available on the CPU.

In the second stage of the reduction only one work-item is launched on one core of the associated CPU. This work-item operates on the array of partial results produced from the previous stage, reducing them sequentially and then outputting the final result.

4.3 Additional Optimisations

Several additional candidate optimisations were subsequently applied to the initial implementation with the aim of further improving performance. Table 3 shows the affect of these optimisations on the overall performance of CloverLeaf on two current but distinctly different target architectures, an Intel Xeon E3-1240 CPU and an Nvidia Tesla K20 (Kepler) GPU.

After the development of the initial implementation performance was particularly poor on CPU-based architectures. This was caused, at least on the Intel architecture, partly by current generations of the Intel OpenCL compilers being unable to successfully vectorise several computationally-expensive kernels. Using the vectorisation reports from the Intel compiler we determined that this was due to a combination of the top-level double loop nest and the array/buffer indexing scheme employed in the initial implementation of these kernels (see Figure 4b). The initial implementation employed a preprocessor macro to map between the original Fortran array index scheme and the underlying OpenCL index scheme. In certain circumstances this caused the initial array index to be set to a negative value before subsequently being mapped back to the appropriate positive index value expected by the OpenCL implementation.

To remedy this situation we subsequently reimplemented the OpenCL device-side kernels, removing the original top-level double loop nest structure (see Figure 4b) and replacing it with a single `if`-test (see Figure 5). This `if`-test prevents grid points from being recalculated or buffers from being accessed beyond their bounds when the kernels are executed with additional work-items due to the `NDRange`-rounding mechanism mentioned previously. We also used OpenCL's `NDRange`-offset facilities to remove the previous array index calculation scheme. This now operates directly in the address range expected by the OpenCL implementations and prevents the index value from being initialised to a negative value. Implementing these optimisations enabled all the kernels within CloverLeaf to be successfully vectorised by the Intel OpenCL compiler involved in this study.

We examined the affect on overall performance of employing the OpenCL preprocessor to replace all constant values within the device-side kernels prior to their compilation. This removed the need to explicitly pass these values into the kernels at run-time via OpenCL's `setArg` mechanism. Additionally we also employed the preprocessor in combination with OpenCL's `NDRange`-offset mechanism to minimise the array/buffer index arithmetic within the kernels.

The in-order command queue used in the initial implementation appropriately captures the dependency chain and synchronisation requirements of the vast majority of kernel invocations within the application. This approach however places unnecessary synchronisation constraints on the kernels at two locations in each timestep: when multiple reduction operations are required in parallel during the *Field Summary* kernel; and when multiple *Update Halo* kernels are launched to operate in parallel on different data buffers. We therefore employed an additional out-of-order command queue to operate alongside the original in-order command queue.

Kernels which can execute in parallel are enqueued in the out-of-order command queue in batches separated by `enqueueBarrier` or `enqueueWaitForEvents` operations which provide synchronisation constructs between these batches. A global event object is used to delay the execution of the first parallel batch of kernels in this queue until the immediate preceding kernel has finished executing within the in-order queue. On particular platforms, however, we found it to be more performant to employ `event-wait` operations between the kernel batches rather than explicitly enqueuing `barrier` operations. We speculate here that on these platforms the enqueuing of the `barrier` operations does not cause the preceding batch of kernels to be executed on the actual target device, but leave the confirmation of this hypothesis to future work.

Our initial implementation relied on the underlying OpenCL runtime system selecting the most appropriate local workgroup size for each kernel invocation, passing `NULL` instead of an `NDRange` to the appropriate argument when each kernel is enqueued. We therefore examined the effect on performance of explicitly specifying the local workgroup size rather than relying on the underlying runtime system.

Additionally we also examined the effect on performance of explicitly calling the OpenCL `flush` operation directly after each kernel is enqueued. With the aim of potentially improving the speed with which kernels are dispatched and executed on the target devices. To facilitate further potential optimisations the `restrict` keyword was also subsequently added to the buffer definitions within each device-side kernel to indicate to the OpenCL compiler that pointer-aliasing is not employed. Similarly, to potentially improve the efficiency with which data values are read back from the buffers within the device-side kernels we also examined the affect of employing the `enqueueMapBuffer` operations instead of the `enqueue[Read|Write]Buffer` operations used in the initial implementation.

Finally we examined whether the technique of merging kernels could deliver any performance benefits for our OpenCL implementation of CloverLeaf. During this optimisation we examined merging kernels at three different places within the overall algorithm/timestep. Specifically merging: the light-weight, predominantly memory copy dominated, *Update Halo* kernels; several more computationally intense *advection* kernels; and also merging the first stage of the *reduction* operations into the immediately preceding kernels.

5. RESULTS

The OpenCL standard guarantees the functional portability of programs. A standards-compliant OpenCL program that works on a GPU will also work on a CPU, however, there is no guarantee of performance. To assess the "performance portability"—whether the same codebase can be performant on many devices—of our OpenCL implementation of CloverLeaf we conducted a series of experiments.

5.1 Experimental Setup

The hardware and software setup used in these experiments is detailed in Tables 1 and 2. In order to fully evaluate the performance and portability of our code, we used a wide range of hardware, including: CPUs from AMD and Intel; GPUs from AMD and Nvidia; and one APU from AMD. All performance results presented show the total application wall-clock time in seconds. Except where noted, all hardware is paired with the corresponding vendor's OpenCL SDK and runtime. Table 1 also lists the flags used to compile the OpenCL kernels, although we only observed these having a minimal effect on performance.

To provide a baseline against which to compare the performance of our OpenCL implementation we also conducted a performance study using, where available, native versions of CloverLeaf optimised for the particular devices. For CPU devices this involved comparing our OpenCL implementation against an optimised OpenMP version of CloverLeaf, which does not currently utilise vector intrinsic operations; and for the Nvidia GPU device, against an optimised CUDA

Version	Tesla K20 (s)	Xeon E3-1240 (s)
Initial version	65.65	477.82[a]/ 1657.59[b]
Remove loops (RL)	61.31	453.58
RL + Out-of-order queue (OoOQ)	56.29	449.46
RL + flush after kernel enqueue (F)	61.30	454.75
RL + Add restrict keyword (R)	61.64	452.22
RL + Preprocessor constants (PC)	59.55	457.69
RL + Fix local workgroup (FLWG)	57.24	451.78
RL + Map/Unmap Buffers (MUmB)	63.59	453.28
RL + Merge Update Halo Kernels (MUHK)	56.76	447.77
RL + Merge two pairs of advection kernels (MAK)	61.35	454.67
RL + FLWG + Merge Reductions into Kernels (MR)	57.17	-
RL + OoOQ + MUHK	53.95	446.18
RL + PC + min array index arithmetic (AIA)	59.76	458.25
RL + OoOQ + PC	54.71	455.16
RL + OoOQ + PC + FLWG	51.84	455.47
RL + OoOQ + PC + FLWG + MUHK	49.16	449.46

[a] Intel OCL 2013
[b] Intel OCL 2012

Table 3: OpenCL optimisations: runtime of the 960^2 problem.

version. Table 1 contains information on the specific OpenMP and CUDA implementations used on each architecture. For AMD GPU devices no such comparison was performed, as OpenCL is the native language on these devices.

CloverLeaf is configured to simulate the effects of a small, high-density region of ideal gas expanding into a larger, low-density region of the same gas, causing a shock-front to form. The configuration is altered by varying the number of cells used in the computational mesh. Finer mesh resolutions increase both the runtime and memory usage of the simulation. In this study we focused on two different problem configurations from the standard CloverLeaf benchmarking suite. We used the 960^2 problem executed for 2955 timesteps to analyse the portability of our OpenCL implementation across a range of hardware devices.

5.2 Optimisations Results Analysis

During the development of our OpenCL implementation of CloverLeaf we analysed the performance of the candidate optimisations outlined in Section 4.3, on two alternative processor architectures, an Intel Xeon E3-1240 CPU and a Tesla K20 (Kepler) GPU from Nvidia. Table 3 presents the results from these optimisation experiments and Table 1 contains more detailed information on the software and hardware setup employed. To assess the success of each of these optimisations we used the 960^2 benchmark problem from the standard CloverLeaf benchmarking suite. This benchmark problem contains approximately 1 million cells, and executes for a simulated 15.5 microseconds (2955 timesteps).

Through these experiments we were able to improve the performance of our initial OpenCL implementation of CloverLeaf by 25.1% on the Nvidia K20 platform, down from 65.56s to 49.16s and from 477.82s to 446.18s (a 6.6% improvement) on the Intel Xeon E3-1240 architecture. Table 3 also shows the performance improvements which Intel have made to their OpenCL compiler and runtime system between the 2012 and 2013 releases. Our initial OpenCL version takes 1657.89s when Intel's 2012 OpenCL SDK is used compared to only 477.82s when Intel's 2013 SDK is employed, an improvement of 71.2%.

Using the reporting capabilities of Intel's offline OpenCL compiler we were able to determine that the compiler was unable to vectorise 23 kernels within the initial version. Implementing the *remove loops* optimisation (see Section 4.3 and Figures 4b and 5) enabled the compiler to successfully vectorise the problematic kernels and reduced the overall runtime of the benchmark by 24.24s (5.1%) on the Xeon CPU architecture. This optimisation was also effective on the Nvidia K20 architecture, reducing the overall runtime by 4.34s (6.6%).

Employing an out-of-order command queue where appropriate within CloverLeaf also proved to be effective, reducing overall runtime by 5.02s (8.2%) and 4.12s (0.9%) respectively on the GPU and CPU architectures compared to the equivalent version which did not employ this technique. Additionally fixing the local workgroup size of the kernels when they are enqueued, rather than letting the OpenCL runtime select an appropriate configuration, also delivered a performance improvement on both architectures. Although the performance improvement was more significant on the Kepler GPU, 4.1s (6.6%) compared to only 1.8s (0.4%) on the Xeon CPU. Using the OpenCL preprocessor to pass constant values into the kernels during compilation rather than at runtime also delivered a performance improvement of 1.76s (2.9%) on the GPU architecture. Surprisingly, however, in our experiments this optimisation actually slightly reduced performance on the CPU platform by 4.1s (0.9%). We are still investigating the reason for this effect and will report on it in future work.

Using the OpenCL `flush` construct immediately after a kernel is enqueued and the `restrict` keyword on the data buffers did not significantly affect performance on either architecture. With the runtimes of versions which employed these optimisations being almost identical to equivalent versions which did not. Similarly using the OpenCL `map` and `unmap` constructs when reading data back from buffer objects also did not affect performance on the CPU architecture, although it did reduce performance on the GPU architecture by 2.3s (3.7%). We believe that this is likely due to this mechanism being less effective than the OpenCL buffer `read` constructs, over the PCI-bus which connects the GPU

Device	OpenCL (s)	Native (s)	Slowdown (%)
Xeon E3-1240	449.46	432.08	4.02
Opteron 6272	798.67	475.59	67.93
Trinity A10-5800K	648.97	-	-
Tesla K20	49.16	42.78	14.91
Firepro V7800	87.39	-	-

Table 4: Runtime of the 960^2 problem.

to the host processor.

We also examined the effect of merging OpenCL kernels within the application. Merging several of the short, computationally light, *Update Halo* kernels delivered a significant performance improvement on both architectures. On the GPU this improved performance by 4.55s (7.4%) and by 5.8s (1.3%) on the CPU architecture. However merging several of the longer, more computationally intense advection kernels did not deliver a significant affect on the overall runtime of the application.

Building on this analysis we subsequently created versions of CloverLeaf which contained the most successful optimisations discussed above. The last 4 lines of Table 3 detail the performance of these versions. On the GPU architecture the most successful version combined: the *remove loops*; the *out-of-order queue*; the *preprocessing of constants*; the *fixing of the local workgroup size* and the *merging of the update-halo kernels* optimisations. This version reduced the overall runtime by 16.5s (25.1%) on the GPU architecture, compared to the initial implementation, and by 28.4s (5.9%) on the CPU architecture. However on the CPU architecture the most performant version only employed: the *remove loops*; the *out-of-order queue* and the *merge update-halo kernels* optimisations. This version reduced the overall runtime of the application by 31.6s (6.6%) compared to the initial version.

5.3 Single-Node Results Analysis

Following our analysis in Section 5.2 we selected the most performant version of CloverLeaf on the K20 GPU architecture to function as the reference version for the remainder of our experiments.

To evaluate the "performance portability" of this version we conducted a series of experiments, again using the 960^2 benchmark problem (executed for 2955 timesteps), on each of the five hardware platforms described in Table 1. The approximate memory usage of this problem is 500MB, meaning it will fit within the available memory on all of the devices employed in this study. We compared the total application wall-time of our OpenCL implementation against the best native version of the code on each particular platform. Figure 4 contains the results of our experiments.

On the CPU-based architectures of the Intel Xeon E3-1240 and the AMD Opteron 6272 our OpenCL implementation was within 17.4s and 323.05s respectively of the optimised native OpenMP implementation. On the GPU-based architectures our implementation took 87.39s on the AMD Firepro V7800 and 49.16s on the Nvidia Tesla K20. The same benchmark problem running on the GPU component of an AMD A10-5800K (AMD's Trinity line of APUs) took 648.97s.

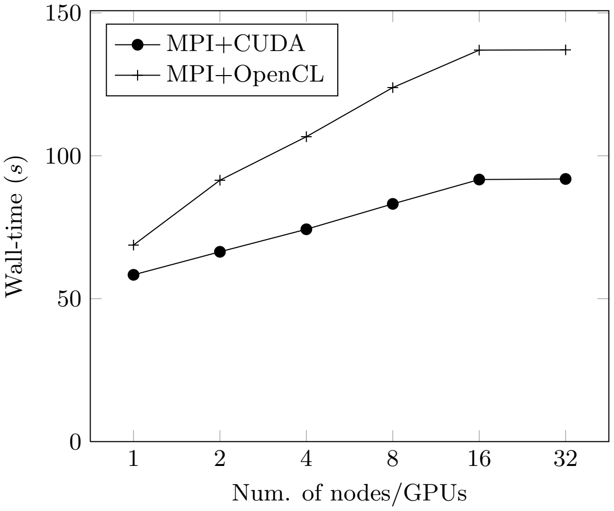

Figure 6: Wall-times for the 960^2 problem weak scaled.

Whilst in all cases the performance of our OpenCL implementation does not quite match the performance of the native implementations. It is within only 4.02% of the performance of the optimised native OpenMP version on the Intel Xeon E3-1240 platform, however, performance on the AMD Opteron 6272 (Interlagos) CPU architecture is more disappointing. On this platform our OpenCL implementation experiences a 67.9% slowdown in performance compared to an equivalent optimised OpenMP version. We believe that this is due, at least partially, to the AMD OpenCL runtime being unable to generate vector instructions unless vector types or operations are explicitly used within the OpenCL code. As we do not currently employ these constructs in our OpenCL implementation, we plan to validate this hypothesis in future work.

On the Nvidia K20 architecture our OpenCL is closer to the performance of the optimised native CUDA implementation, experiencing only a 6.4s (14.91%) drop in performance.

5.4 Multi-Node Results Analysis

To fully assess OpenCL's suitability as a candidate technology for implementing the hybrid programming model, we integrated our OpenCL implementation of CloverLeaf with the existing MPI implementation. In this new implementation OpenCL is used for all intra-node computation and communication, and MPI is utilised for all inter-node communication.

To assess the performance of our OpenCL implementation we conducted a scaling study which examined the performance of the code when applied to the standard 960^2 cell problem from the CloverLeaf benchmarking suite. We weak scaled this particular problem class on the Cray XK6 platform described in Table 2, from 1 GPU up to 32 GPUs. To provide a baseline against which to compare the performance and scalability of our OpenCL implementation, we also conducted an equivalent weak scaling study of the same problem using an optimised CUDA based version of CloverLeaf.

The results from both of these scaling experiments can be found in Figure 6. This demonstrates that whilst the performance of our MPI+OpenCL based implementation is not able to match that of the optimised native MPI+CUDA version it is able to demonstrate broadly similar scaling behaviour. Although the initial scalability is slightly worse at lower node counts both versions follow a very similar trend in the experiments on higher numbers of nodes/GPUs.

5.5 Portability Analysis

Although the performance of our OpenCL implementation is not able to quite match that of the native versions of the code, it does exhibit significantly improved portability across a range of architectures. The CUDA and OpenMP versions of the code are effectively confined to Nvidia GPUs and CPU devices respectively. This improved portability offered from a single code base for, in some cases, only a relatively small performance penalty may be an extremely attractive trade-off for HPC sites as they attempt to cope with ever increasing workloads and a myriad of complex programming models and architectures.

6. CONCLUSIONS

As the heterogeneity of architectures and levels of on-node parallelism increase in the approach to the era of exascale computing, harnessing the power of both host and accelerator devices will become critically important. Memory constraints are also limiting the scalability of pure-MPI applications, and alternative hybrid programming approaches must be considered.

The results presented here demonstrate that OpenCL provides a viable mechanism for achieving portable performance in scientific codes. Using OpenCL for CloverLeaf, an explicit Lagrangian-Eulerian hydrodynamics application, we have extended the promise of functional portability to one of performance portability. On certain platforms our OpenCL based implementation is only 4% slower that an equivalent optimised native implementation, although it can be as much as 68% slower on others. It is also extremely likely that this performance gap will shrink even further as more optimisations are implemented as the development of CloverLeaf continues.

Our OpenCL implementation has the added advantage of exhibiting significantly superior portability when compared to the alternative implementations of CloverLeaf. It is the only implementation which, from a single code-base, is able to execute across the diverse range of current HPC architectures involved in this study. This increased portability may well be an extremely attractive trade-off against raw performance for HPC sites as they struggle to cope with increasingly complex programming models and architectures. We have also shown that OpenCL is a viable technology for combining with MPI in order to implement the hybrid programming model.

Additionally this work has demonstrated that developing OpenCL kernels and codes in particular ways is critical for achieving good performance. We found that removing loop structures; employing out-of-order command queues; merging kernels; explicitly specifying a local workgroup size and using the OpenCL preprocessor for constant values particularly key to improving performance.

In future work we plan to apply further optimisations to our OpenCL implementation of CloverLeaf, including investigating the effect of using explicit vector types and operations. In particular we plan to examine the cause of the large slowdown we experienced on the AMD Interlagos platform and explore potential optimisations for this architecture. We also intend to examine in more detail the affect of setting different local workgroup sizes for the various OpenCL kernels within CloverLeaf. We feel that this may be a suitable area for employing an auto-tuning mechanism to automatically select an optimal configuration. Additionally we aim to examine approaches for merging more of the kernels and utilising one dimensional arrangements of work-items to process multiple grid points per work-item.

We also plan to investigate a more advanced hybrid model, in which the CPU does not act as merely a host, but shares some of the computational work with the attached accelerator device. This may prove particularly effective on integrated CPU-GPU devices such as AMD APUs, on which we also plan to evaluate the effect of the "zero-copy" OpenCL constructs.

To further assess OpenCL's suitability as a technology for implementing the hybrid programming model. We plan to analyse the overall memory consumption of our implementation compared to alternative approaches and explore whether explicitly packing the MPI communication buffers on the attached GPU devices can deliver any performance advantages compared to the OpenCL built-in functions.

With the aim of further improving the reduction operations within CloverLeaf, we intend to evaluate the recently released ArrayFire software from Accelereyes [6] and to evaluate the improvements made to the atomic operations on Nvidia's new Kepler architecture. We also intend to further optimise our reduction kernels and incorporate them into an optimised library which others can use for their own OpenCL projects.

Finally, we hope to evaluate the performance of the OpenCL version of CloverLeaf on a wider range of platforms, including Intel Xeon Phi products, Altera FPGA devices, and ARM-based APUs such as the Samsung Exynos 5250. Additionally, we also intend to examine how best to execute OpenCL codes across multi-CPU nodes which contain numerous NUMA (non-Uniform Memory Access) regions and to investigate the effect device fissioning has upon performance on these platforms.

7. ACKNOWLEDGMENTS

This work is supported by the UK Atomic Weapons Establishment under grants CDK0660 (The Production of Predictive Models for Future Computing Requirements) and CDK0724 (AWE Technical Outreach Programme) and also the Royal Society through their Industry Fellowship Scheme (IF090020/AM).

The authors would like to express their thanks to Sandia National Laboratories for access to the Teller Testbed system. Sandia National Laboratories is a multi-programme laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration (NNSA) under contract DE-AC04-94AL85000. Teller is provided under the United States Department of Energy's ASCR and NNSA Testbed Computing Programme.

We would also like to thank John Pennycook of the University of Warwick for his help and advice on OpenCL. Additionally we would also like to thank Simon McIntosh-Smith and Michael Boulton of the University of Bristol for developing the CUDA implementation of CloverLeaf and NVIDIA Corporation for their optimisations to the CUDA version.

CloverLeaf is available via GitHub (see `http://warwick-pcav.github.com/CloverLeaf/`), and is also released as part of the Mantevo project hosted at Sandia National Laboratories which can be accessed at `https://software.sandia.gov/mantevo/`.

8. REFERENCES

[1] The OpenCL Specification version 1.2. `http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`, November 2011.

[2] Accelerated Parallel Processing (APP) SDK. `http://developer.amd.com/tools/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/`, November 2012.

[3] Intel SDK for OpenCL Applications 2012. `http://software.intel.com/en-us/vcsource/tools/opencl-sdk`, November 2012.

[4] OpenCL Lounge. `https://www.ibm.com/developerworks/community/alphaworks/tech/opencl`, November 2012.

[5] OpenCL NVIDIA Developer Zone. `https://developer.nvidia.com/opencl`, November 2012.

[6] ArrayFire. `http://www.accelereyes.com`, January 2013.

[7] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, and J. Träff. MPI on a Million Processors. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2009.

[8] B. Bergen, M. Daniels, and P. Weber. A Hybrid Programming Model for Compressible Gas Dynamics using OpenCL. *39th International Conference on Parallel Processing Workshops*, 2010.

[9] R. Brook, B. Hadri, V. Betro, R. Hulguin, and R. Braby. Early Application Experiences with the Intel MIC Architecture in a Cray CX1. In *Cray User Group*, 2012.

[10] W. Brown, P. Wang, S. Plimpton, and A. Tharrington. Implementing molecular dynamics on hybrid high performance computers - short range forces. *Computer Physics Communications*, 182(4), 2011.

[11] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J. Andre, D. Barkai, J. Berthou, T. Boku, B. Braunschweig, et al. The International Exascale Software Project roadmap. *International Journal of High Performance Computing Applications*, 25(1), 2011.

[12] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, 38(8), 2010.

[13] J. Fang, A. Varbanescu, and H. Sips. A Comprehensive Performance Comparison of CUDA and OpenCL. In *Proceedings of the 40th International Conference of Parallel Processing*, 2011.

[14] X. Gao, Z. Wang, H. Wan, and X. Long. Accelerate Smoothed Particle Hydrodynamics Using GPU. In *Proceedings of the 2nd IEEE Youth Conference on Information Computing and Telecommunications*, 2010.

[15] M. Harris. Optimizing Parallel Reduction in CUDA. `http://docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pdf`, accessed Jan 2013.

[16] J. Herdman, W. Gaudin, S. McIntosh-Smith, M. Boulton, D. Beckingsale, A. Mallinson, and S. Jarvis. Accelerating Hydrocodes with OpenACC, OpenCL and CUDA. In *Proceedings of the 2rd International Workshop on Performance Modelling, Benchmarking and Simulation*, 2012.

[17] J. Junior, E. Clua, A. Montenegro, and P. Pagliosa. Fluid simulation with two-way interaction rigid body using a heterogeneous GPU and CPU environment. In *Proceedings of the Brazilian Symposium on Games and Digital Entertainment*, 2010.

[18] R. Karrenberg and S. Hack. Improving Performance of OpenCL on CPUs. In *Proceedings of the 21st International Conference on Compiler Construction*, 2012.

[19] J. Kim, S. Seo, J. Lee, J. Nah, and J. Jo, G. Lee. SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters. In *Proceedings of the 26th ACM International Conference on Supercomputing*, 2012.

[20] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi. Evaluating Performance and Portability of OpenCL Programs. In *Proceedings of the 5th International Workshop on Automatic Performance Tuning*, 2010.

[21] S. Pennycook, S. Hammond, S. Wright, J. Herdman, I. Miller, and S. Jarvis. An investigation of the performance portability of OpenCL. In *Journal of Parallel Distributed Computing*, 2012.

[22] S. Pennycook and S. Jarvis. Developing Performance-Portable Molecular Dynamics Kernels in OpenCL. In *Proceedings of the 2rd International Workshop on Performance Modelling, Benchmarking and Simulation*, 2012.

[23] J. Phillips, J. Stone, and K. Schulten. Adapting a Message-Driven Parallel Application to GPU-Accelerated Clusters. In *Proceedings of the 20th International Conference for High Performance Computing, Networking, Storage and Analysis*, 2008.

[24] J. Pier, I. Figueroa, and J. Huegel. CUDA-enabled Particle-based 3D Fluid Haptic Simulation. In *Proceedings of the Electronics, Robotics and Automotive Mechanics Conference*, 2011.

[25] E. Rustico, G. Bilotta, G. Gallo, and A. Herault. Smoothed Particle Hydrodynamics Simulations on Multi-GPU Systems. In *Proceedings of the 20th*

Euromicro International Conference on Parallel, Distributed and Network-Based Processing, 2012.

[26] H. Shukla, T. Woo, H. Schive, and T. Chiueh. Multi-Science Applications with Single Codebase - GAMER - for Massively Parallel Architectures. In *Proceedings of 23rd International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[27] T. Stefanski, N. Chavannes, and N. Kuster. Hybrid OpenCL-MPI parallelization of the FDTD method. In *Proceedings of the 13th International Conference on Electrmagnetics in Advanced Applications*, 2011.

[28] J. van der Sanden. Evaluating the Performance and Portability of OpenCL. `http://alexandria.tue.nl/extra1/afstversl/wsk-i/sanden2011.pdf`, 2011.

[29] R. Weber, A. Gothandaraman, R. Hinde, and G. Peterson. Comparing Hardware Acclerators in Scientific Applications: A Case Study. *IEEE Transactions on Parallel and Distributed Systems*, 22(1), 2010.