INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN FLUIDS
Int. J. Numer. Meth. Fluids 2011; **00**:1–18
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/fld

Finite element assembly strategies on multi- and many-core architectures

G. R. Markall[1]*, A. Slemmer[1], D. A .Ham[2], P. H. J. Kelly[1], C. D. Cantwell[3], and S. J. Sherwin[3]

[1]*Department of Computing, Imperial College London*
[2]*Department of Earth Science & Engineering and Grantham Institute for Climate Change, Imperial College London*
[3]*Department of Aeronautics, Imperial College London*

SUMMARY

We demonstrate that radically differing implementations of finite element methods are needed on multi-core (CPU) and many-core (GPU) architectures, if their respective performance potential is to be realised. Our experimental investigations using a finite element advection-diffusion solver show that increased performance on each architecture can only be achieved by committing to specific and diverse algorithmic choices that cut across the high-level structure of the implementation. Making these commitments to achieve high performance for a single architecture leads to a loss of performance portability.
Data structures that include redundant data but enable coalesced memory accesses are faster on many-core architectures, whereas redundancy-free data structures that are accessed indirectly are faster on multi-core architectures. The *Addto* algorithm for global assembly is optimal on multi-core architectures, whereas the *Local Matrix Approach* is optimal on many-core architectures despite requiring more computation than the Addto algorithm. These results demonstrate the value in making the correct choice of algorithm and data structure when implementing finite element methods, spectral element methods and low-order discontinuous Galerkin methods on modern high-performance architectures. Copyright © 2011 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Finite element method; GPU; Multicore; Manycore

1. INTRODUCTION

The increasing use of many-core architectures (such as AMD and NVidia GPUs) for scientific computations prompts investigation to determine the optimal implementation strategies on these devices. Although significant speedups have been demonstrated for finite element applications [1, 2, 3, 4, 5], it is necessary to spend a considerable amount of time and effort tuning the code in order to obtain optimal performance.

It is common to perform manual optimisations to increase the performance of code at a local level in sequential implementations of scientific codes. These transformations include loop unrolling, loop tiling, common subexpression elimination, and software pipelining, amongst many other well-documented transformations [6]. In many cases a compiler can perform some of these transformations automatically, but it is still common for scientists to hand-optimise performance-critical sections of their code.

Although these optimisations often result in significant performance gains, the fundamental differences in the design of many-core architectures dictate that taking sequential codes and performing traditional optimisations will not be sufficient to obtain the best performance. Instead, the investigation of more radical code transformations, which can be expressed in terms of transformations of the algorithm, become necessary.

The *global assembly phase* is a performance bottleneck in many finite element applications. Investigations of alternative implementations of global assembly algorithms on CPUs [7, 8, 9] show that depending on problem parameters, the point at which it is profitable to switch algorithms for global assembly varies depending on the problem. We have chosen to focus on the implementation of the global assembly phase as it is a portion of the computation that forms a significant bottleneck, is difficult to optimise on manycore architectures, and is used in a wide variety of finite element implementations.

In this paper we present finite element implementations that are written in both CUDA [10], for NVidia GPUs, and OpenCL [11], which can be compiled to run on AMD and NVidia GPUs and Intel/AMD CPUs. Using these implementations, we show that the point at which it becomes profitable to switch algorithms differs between target architectures. Using OpenCL achieves functional portability across different multicore and GPU platforms; we demonstrate that performance portability is more challenging to achieve.

1.1. Contributions

The contributions of this paper are as follows:

- The main contribution is a thorough mapping of the implementation space of existing algorithms for global matrix assembly in low-order finite element methods on CPU and GPU architectures. This contribution demonstrates that different choices that are made at a high and abstract level are more efficient when implementing finite element methods on different architectures.
- We demonstrate high-performance implementations of a finite element advection-diffusion solver written using CUDA and OpenCL. These implementations give an order of magnitude speedup on an NVidia GTX480 when compared to an optimised baseline implementation running on a 4-core Intel Xeon E5620 Westmere CPU.

This work builds upon our previous contribution [12], in which a proof-of-concept implementation of global assembly algorithms using CUDA was presented. The more extensive explorations presented in this paper can be used by finite element practitioners to influence the development of their codes. These investigations are relevant to the wider community of practitioners implementing high order finite element methods [13], spectral/hp-type methods [14], and low/high order Discontinuous Galerkin methods [15].

1.2. Structure

We begin by explaining the background and motivation for this work (Section 2), and continue with a description of the target architectures that were used in the investigations (Section 3). This frames our discussion of the finite element method and issues in implementing it on these architectures (Section 4). Our implementations of an advection-diffusion solver with consideration for these issues are discussed in Section 5. We present and analyse the performance results of these simulations in Section 6. Finally, we discuss related work in Section 7 and conclude in Section 8.

2. BACKGROUND AND MOTIVATION

The FEniCS project [16] has shown that the *Unified Form Language* (UFL) provides an appropriate level of abstraction of the finite element method for generating efficient code from maintainable sources. Using UFL for writing finite element codes (as opposed to writing it in a low-level language such as C++ for Fortran) is desirable as it prevents common errors and eliminates many time-consuming and error-prone tasks required when developing in a low-level language. UFL allows the user to declaratively specify the computations in a finite element discretisation instead of writing an imperative program that performs these computations. Since this specification does not commit to low-level implementation-specific decisions such as the format of data structures or the

DOI: 10.1002/fld

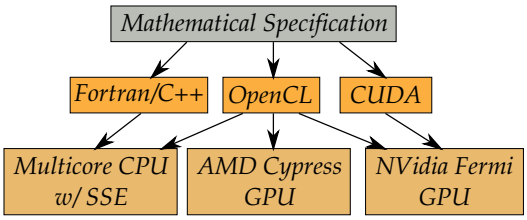

Figure 1. A mathematical specification of a finite element method written in UFL can be compiled into different languages to enable execution on different targets. Although OpenCL is functionally portable across architectures, it is not performance-portable.

particular algorithm used to evaluate an expression, we have been exploring how unmodified UFL sources can be translated into lower-level code for multiple hardware platforms with target-specific optimisations (see Figure 1).

Our research into the generation of code for multiple architectures from UFL sources necessitates the exploration of various implementation spaces in order to gain insight into the optimisations that should be implemented within the code generator. In this paper we present the results from our exploration of one of the design spaces that can be opened up by starting from a high level specification such as UFL.

3. TARGET PLATFORMS

We can characterise the fundamental difference between multi-core and many-core architectures as follows:

Manycore architectures sacrifice the sequential performance of a single core within the processor in order to increase the parallel throughput for streaming workloads. As a result, the many-core design consists of many very simple processing units, very small caches, and a high-bandwidth interface to multiple banks of memory.

Multicore architectures sacrifice the parallel computational throughput of the entire processor in order to increase the performance of single cores for low-latency computation. These goals lead to a design consisting of a few highly complex cores with large caches.

3.1. NVidia Fermi

NVidia Fermi [17] is a representative instance of the many-core design. It is a highly-parallel architecture made up of many minimally complex processing elements which are specialised to perform arithmetic operations. See Figure 2 for an overview of the entire processor. The processor consists of up to 16 *streaming multiprocessors* (SMs), all sharing a common level 2 cache and interface to main memory.

The architecture of an SM is outlined in Figure 3. The SM consists of 32 *streaming processors* (SPs). Each SP executes instructions on integer or floating-point operands. Two SMs are required for computation in double-precision, effectively halving the maximum throughput. The streaming processors also share a register file to support the execution of a large number of concurrent threads.

A distinguishing feature of the SM, and common to manycore architectures in general, is the presence of a software-controlled cache, the *shared memory*. Shared memory can be explicitly programmed to store temporary data that does not fit inside registers, without needing to transfer data to and from the much slower main memory. In Fermi, a total of 64KB of memory is available to each SM, that can be divided up into 16KB/48KB spaces for the shared memory and a level 1 cache.

DOI: 10.1002/fld

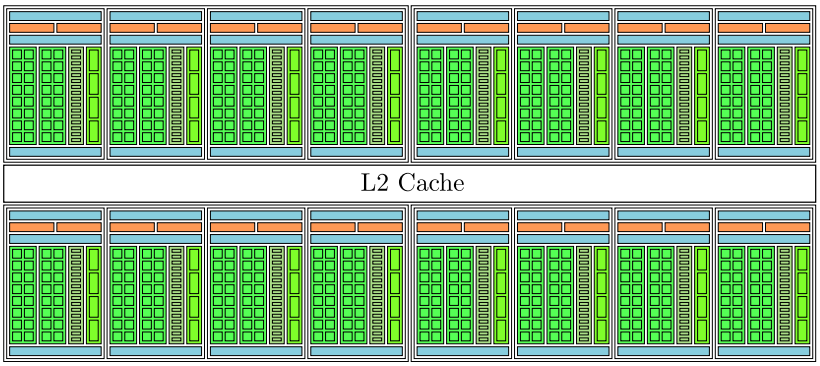

Figure 2. The Fermi Architecture [17]. Up to 16 Streaming Multiprocessors are on die, sharing a common L2 cache and interface to main memory.

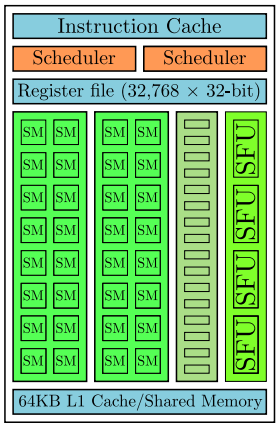

Figure 3. A single Streaming Multiprocessor [17]. 32 Streaming processors share a register file and shared memory. Four special function units (not considered here) compute transcendental functions at a reduced accuracy.

3.2. The Many-core Parallel Execution Model

Many-core processors do not operate independently of the main processor of a machine. The CPU of a machine is responsible for transferring input data into the memory of a many-core device, launching *kernels* on the many-core device, and fetching the output data once computation is complete.

A kernel is a single function call executed on the many-core device in parallel by all of the cores. The programming model for kernels allows work (usually data-parallel operations) to be divided between a large number of *threads*. In Fermi, threads are grouped at several granularities. A *warp* is a group of 32 threads that all share the same program counter, and as a result must all execute the same instruction concurrently. The next level of granularity is the *block*, which is made up of several warps. Each block is mapped onto a particular SM, and has an affinity to that SM for the lifetime of the kernel. No communication between threads in different blocks is allowed to take place. During the execution of a single kernel, one *grid* exists, which contains all the blocks. This organisation of threads and the lack of communication between blocks determines the strategies that are used to obtain good performance on a many-core architecture, which we describe in the next subsection.

3.3. Performance Considerations

We highlight the main performance considerations that must be taken into account when writing code for many-core architectures:

DOI: 10.1002/fld

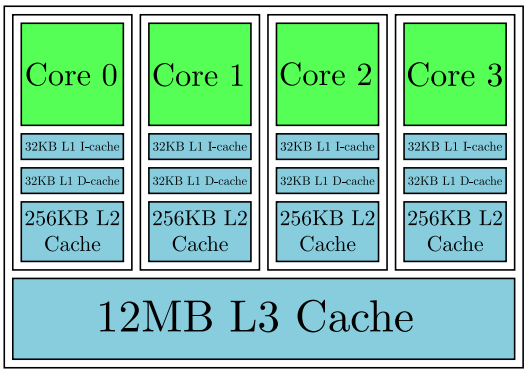

Figure 4. The Nehalem architecture. A small number of complex cores have large L2 caches and share a very large L3 cache.

- A massive amount of fine-grained parallelism is required in order to make full use of all the cores on a GPU. This is typically rendered to the user as a requirement to launch thousands of concurrent threads.
- Because many threads run concurrently on a single multiprocessor, the per-thread state must be extremely small as it is stored within registers and a very small amount of cache. More registers per thread reduces the maximum number of running threads, reducing the scope to hide memory access latency. Additionally the limited cache resources make the exploitation of temporal locality difficult.
- Spatial data locality is required across the threads in a warp, as *coalesced* memory access is needed for high memory bandwidth utilisation. Coalescing is achieved when vectors of threads concurrently access data within a certain-sized (typically 64 or 128 bytes) memory window.
- Performance is also dependent on spatial branch locality across the threads in a warp. Because warps all share a single program counter, they execute the same code path concurrently. When threads within a warp take different paths, execution is serialised between these two paths, reducing performance.
- As GPU memory is separate from the main memory of a machine, data must be transferred to and from it before and after execution. Because of this overhead it is important that a large enough workload is provided in order to benefit from GPU performance.

3.4. Multicore architectures

Since multicore architectures are very similar to the single-core CPU architectures that are ubiquitous in high-performance computing, we avoid giving an exhaustive description of their design. Instead, we contrast the design of multicore architectures with that of manycore architectures, and briefly describe how these differences imply a different strategy for writing efficient code. Figure 4 gives a schematic overview of Intel's Nehalem architecture, which is a recent instance of a multicore architecture and is used in our experiments. The key differences in multicore architectures include:

Core design and count. Multicore CPUs consist of a small number of highly complex cores. Each core is optimised for executing serial workloads with very low latency.

Cache size. The large caches supporting each core assist in the reduction of latency. The caches are hardware controlled, requiring no effort on behalf of the programmer.

Memory bandwidth. The large caches in a multicore architecture hide a lower (than in many-core) main memory bandwidth.

Instead of decomposing the workload into thousands of small data-parallel tasks, the usual implementation strategy on a multicore architecture is to partition the workload in a coarse-grained

DOI: 10.1002/fld

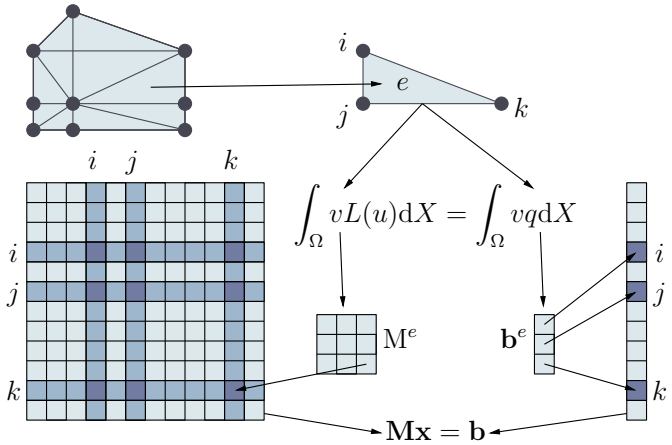

Figure 5. Overview of the computations performed in the finite element method. Elements are looped over and integrals evaluated over them to produce the local matrices and vectors, $\mathbf{M}^e$ and $\mathbf{b}^e$. These are added into a sparse global matrix and a global vector that form a system of equations.

manner. For example in finite element assembly, one may choose to partition the mesh into several regions, assigning one region to each core. Each core would perform assembly for its region independently. By contrast, it would be more usual on a manycore architecture to decompose the domain into individual elements, and assigning one element to each thread. In the following section, we discuss these implementation choices more extensively.

4. THE FINITE ELEMENT METHOD

The finite element method is used for discretising the weak form of partial differential equations. Here, in order to establish our terminology and scope, we provide an overview of the computations that are performed in the finite element assembly and solution process (see Figure 5). We highlight the key data structures and how they are accessed, since this affects how the code is mapped onto the different targets. For a complete treatment of the method, see [14]. Solving a partial differential equation with a time-varying solution using the finite element method typically consists of the following phases for each timestep:

Local Assembly: For each element e in the domain, an $N \times N$ matrix, $\mathbf{M}^e$, and an N-length vector, $\mathbf{b}^e$, are computed, where N is the number of nodes per element. These are referred to as *local* matrices and vectors. Computing these matrices and vectors usually involves the evaluation of integrals over the element using Gaussian quadrature. Meshes are typically unstructured, requiring the use of indirection to gather together the required data associated with each element. In many implementations, every element has the same number of nodes, but this is not required by the method.

Global Assembly: The local matrices, $\mathbf{M}^e$, and vectors, $\mathbf{b}^e$, are used to form a *global matrix*, $\mathbf{M}$, and *global vector*, $\mathbf{b}$, respectively. This process couples the contributions of elements together. The sparsity structure of the global matrix depends on the connectivity of the mesh, and as such it is typically very sparse. The Compressed Sparse Row (CSR) [18] format is often used to reduce the storage requirement of the matrix and to eliminate redundant computations.

Solution: The system of equations $\mathbf{Mx} = \mathbf{b}$ is solved for $\mathbf{x}$, often using an iterative method, which requires computation of the *sparse matrix-vector product* (SpMV) $\mathbf{y} = \mathbf{Mv}$.

We shall examine the global assembly phase as it often accounts for a large proportion of the execution time. This phase consists of performing the following computations:

DOI: 10.1002/fld

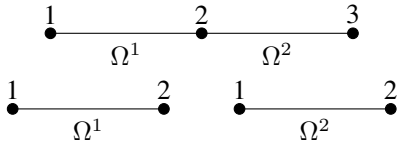

Figure 6. *Top:* A 1D domain decomposed into two elements (Ω^i). *Bottom:* Local node numbering of individual elements.

$$\mathbf{M} = \mathcal{A}^T \mathbf{M}^E \mathcal{A} \tag{1}$$

$$\mathbf{b} = \mathcal{A}^T \mathbf{b}^E \tag{2}$$

where $\mathcal{A}$ is a matrix mapping the local node numbers of each element to the global node numbers, $\mathbf{M}^E$ is a block-diagonal matrix whose e-th block is $\mathbf{M}^e$, and $\mathbf{b}^E$ is a vector of stacked $\mathbf{b}^e$. We shall examine algorithms that can be used to implement these computations, as the optimal choice of algorithm also depends on the target hardware. Consider a two-element, three-node decomposition of a 1-dimensional domain (see Figure 6). In this example, the matrices and vector are:

$$\mathcal{A} = \begin{bmatrix} 1 & & \\ & 1 & \\ & 1 & \\ & & 1 \end{bmatrix} \qquad \mathbf{b}^E = \begin{bmatrix} b_1^1 \\ b_2^1 \\ b_1^2 \\ b_2^2 \end{bmatrix} \tag{3}$$

$$\mathbf{M}^E = \begin{bmatrix} m_{11}^1 & m_{12}^1 & & \\ m_{21}^1 & m_{22}^1 & & \\ & & m_{11}^2 & m_{12}^2 \\ & & m_{21}^2 & m_{22}^2 \end{bmatrix} \tag{4}$$

where m_{ij}^e is the i,j-th term of $\mathbf{M}^e$, and b_i^e is the i-th term of $\mathbf{b}^e$. The structure of $\mathcal{A}$ arises from the geometry of the elemental decomposition of the domain.

It is often inefficient to compute the matrix multiplications described in Equations 1 and 2 on traditional architectures due to the sparsity of $\mathcal{A}$. The *Addto* algorithm is usually more efficient. To describe this algorithm, we first define an array, $\texttt{map}[e][i]$, that maps the local node i of the element e to a global node number. In our example, the array is defined as:

$$\texttt{map}[1][i] = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \quad \texttt{map}[2][i] = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

Figures 7 and 8 describe the *Addto* algorithm for computing $\mathbf{M}$ and $\mathbf{b}$. Intuitively, terms of the local matrix (or vector) of each element are summed into particular terms in the global matrix (or vector) depending on the node numbers of the element. We note that the loop nest that implements the algorithm relates to the algebra from which it is derived - in particular it performs the gather described by the structure of $\mathcal{A}$ in Equation 1. The implementation of this algorithm can be a pinch point for performance on some architectures. For detailed investigations into its performance on CPU architectures, see [7, 8, 9].

4.1. Implementation on GPUs

These algorithms are massively data-parallel, as iterations of all the loops can be executed independently of others. Although this appears to make them ideal for implementing on GPU architectures, there are two issues. First, data races occur if threads concurrently update the same term of the global matrix. Costly atomic operations (or colouring) must be used to ensure correctness. Second, $\mathbf{M}$ is often stored using a format such as *compressed sparse row* (CSR).

DOI: 10.1002/fld

$\mathbf{M} = \mathbf{0}$
for each Element e **do**
  for $i \leftarrow 1$ to N **do**
    for $j \leftarrow 1$ to N **do**
      $\mathbf{M}[\texttt{map}[e][i], \texttt{map}[e][j]]$+=$\mathbf{M}^e[i, j]$
    end for
  end for
end for

Figure 7. Addto for global matrix assembly.

$\mathbf{b} = \mathbf{0}$
for each Element e **do**
  for $i \leftarrow 1$ to N **do**
    $\mathbf{b}[\texttt{map}[e][i]]$+=$\mathbf{b}^e[i]$
  end for
end for

Figure 8. Addto for global vector assembly.

Finding the location in memory of a particular term requires a bisection search of the sparsity structure of the matrix, leading to uncoalesced accesses and control flow divergence within warps (see Section 3).

To avoid these issues, we can derive an alternative algorithm, referred to as the *Local Matrix Approach* (LMA) [7], noting that the only use of $\mathbf{M}$ is for computation of the product $\mathbf{Mv}$ in the solution phase. We omit the global assembly of $\mathbf{M}$ (Equation 1) altogether, and when computation of $\mathbf{y} = \mathbf{Mv}$ is required, the following computation is performed:

$$\mathbf{y} = \left(\mathcal{A}^T \left(\mathbf{M}^E \left(\mathcal{A}\mathbf{v}\right)\right)\right) \tag{5}$$

It is not possible to avoid the assembly of $\mathbf{b}$, as it is explicitly required by the solver. However, we can eliminate the use of atomic operations by computing the matrix-vector product $\mathbf{b} = \mathcal{A}^T\mathbf{b}^E$ using an SpMV kernel instead of using the Addto algorithm.

We note that using the Local Matrix Approach instead of the Addto algorithms results in an increase in computation and memory bandwidth usage in the solver phase proportional to the average number of elements that share a single node (the *variance*) of the mesh. However, its implementation avoids the use of atomic operations and bisection searches in the global assembly phase. In [7, 8, 9], the optimal choice of algorithm in CPU implementations depends on problem parameters, such as the polynomial order of the approximation. We demonstrate in Section 5 that the optimal choice of algorithm depends on the target hardware.

4.2. Colouring

Colouring can be used to avoid the need for atomic operations in the implementation of the global assembly phase. Data races are avoided by assigning each update to the same row of the matrix a different colour. The Addto kernel is then invoked for each colour sequentially. Colouring can be required in an OpenCL implementation, as it does not support atomic operations on double-precision floating-point values.

Performing the colouring reduces the total available parallelism, as only updates of the same colour can execute in parallel. However, in the implementation of the Addto algorithm, the maximum number of colours is equal to the maximum variance of the mesh. Typically, up to 10 colours are required with a 2D mesh, and up to 30 with a 3D mesh. As there will typically be hundreds of thousands of rows in the global matrix, there is still a large amount of parallelism available within each colour. It has been shown in experiments described in [5] that varying the

DOI: 10.1002/fld

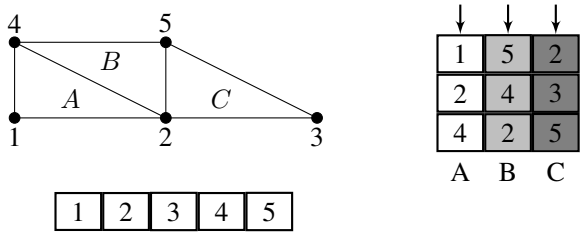

Figure 9. *Top:* A 2D domain decomposed into three elements. *Bottom:* Node data layout in CPU implementation. *Right:* Node data layout in GPU implementation. Threads accessing data in different elements (arrowed) achieve coalescing.

number of colours between 13 and 26 on the same mesh affects the runtime of the assembly phase by approximately 7.5% between the best and worst cases.

We use the structure of the matrix $\mathcal{A}$ in order to determine an optimal colouring of the rows of the local matrices. The first nonzero in each column is tagged with colour 0, and the next non-zero in each column is tagged with the colour 1, and so on until all of the non-zeroes of the matrix are tagged. Subsequently, the rows of each local matrix corresponding to the elements of $\mathcal{A}$ that are tagged with the same number are placed into the same work set, since none of them can conflict. The Addto kernel is invoked separately for each work set, avoiding the need for inter-block synchronisation that would be needed to prevent work on one set beginning before the previous set is finished. For the matrix $\mathcal{A}$ for the domain described earlier, the tagging and work sets are as follows:

$$\mathcal{A} = \begin{bmatrix} 1_0 & & \\ & 1_0 & \\ & 1_1 & \\ & & 1_0 \end{bmatrix} \tag{6}$$

$$Set_0 = \{m^1_{1,*}, m^1_{2,*}, m^2_{2,*}\} \tag{7}$$

$$Set_1 = \{m^2_{1,*}\} \tag{8}$$

We see that no element in a work set conflicts with another element in the same set - in this case, the conflict between $m^1_{2,*}$ and $m^2_{1,*}$ is avoided by placing them in different work sets.

4.3. Data formats

In general, data structures must be carefully chosen to achieve optimal performance (e.g. for cache-optimality on a CPU), and the optimal choice of data structure depends on characteristics of the target architecture. In order to examine the structures that can be used when implementing the finite element method on CPUs and GPUs, we consider a three element domain (see Figure 9).

In CPU implementations, nodal data is often stored on a per-node basis. When data for the nodes of a single element is needed, the mapping array (`map`) is used to indirectly access the nodal data. Although this can lead to poor cache performance due to random access into the nodal data structure, reordering optimisations can be used to minimise this overhead.

This data format is inefficient for GPU implementations, where coalesced accesses must be used to maximise memory performance. It is difficult to achieve coalesced access because the nodal data structure is accessed in a somewhat random fashion. We propose that it can be more efficient to store nodal data on a per-element basis in GPU implementations, interleaving the nodal data for each node of each element. This leads to some redundancy in the storage of nodal data, again proportional to the average variance of nodes; however, it allows coalesced accesses when there is a one-to-one mapping between threads and elements.

Prepared using fldauth.cls
DOI: 10.1002/fld

4.4. Implementation of LMA

The Local Matrix Approach is implemented by considering the computation in Equation 5 in three stages:

$$\underbrace{\mathbf{t} = \mathcal{A}\mathbf{v}}_{\text{Stage 1}}, \qquad \underbrace{\mathbf{t}' = \mathbf{M}^E\mathbf{t}}_{\text{Stage 2}}, \qquad \underbrace{\mathbf{y} = \mathcal{A}^T\mathbf{t}'}_{\text{Stage 3}}. \tag{9}$$

Since $\mathcal{A}$ contains only one non-zero entry per row that is always 1, Stage 1 is implemented as a gather. This involves uncoalesced memory accesses but is more efficient than using an SpMV kernel. The implementation of Stage 2 exploits the block-diagonal structure of $\mathbf{M}^E$ to achieve coalesced accesses and maximal reuse of matrix values. Stages 1 and 2 are implemented in a single kernel. Stage 3 is implemented as an SpMV kernel that is optimised for all the non-zero values equalling 1. Because a global barrier is required between Stages 2 and 3, Stage 3 is implemented in a separate kernel.

5. EXPERIMENTS

We evaluate the performance of the Addto algorithm and the Local Matrix Approach on GPUs using an implementation of a test problem that solves the advection-diffusion equation:

$$\frac{\partial T}{\partial t} + \nabla \cdot (\mathbf{u}T) = \nabla \cdot \overline{\overline{\mu}} \cdot \nabla T \tag{10}$$

where T is the concentration of a tracer, t is time, $\mathbf{u}$ is velocity, and $\overline{\overline{\mu}}$ is a rank-2 tensor of diffusivity. This problem is chosen as it is both a sub-problem and simplified model of a full computational fluid dynamics system. To describe the discretisation of the scheme, we first define the following bilinear forms:

$$m(v,T) = \int_\Omega vT \,\mathrm{d}X \tag{11}$$

$$d(v,T) = \Delta t \int_\Omega \nabla v \cdot \overline{\overline{\mu}} \cdot \nabla T \,\mathrm{d}X \tag{12}$$

$$a(v,T) = \Delta t \int_\Omega \nabla v \cdot \mathbf{u}T - vT\nabla \cdot \mathbf{u} \,\mathrm{d}X \tag{13}$$

where Δt is the timestep size and Ω is the domain. A split scheme is used, first solving for advection using a finite element spatial discretisation and a 4th order Runge-Kutta temporal discretisation:

$$m(v,T_1) = a(v,T^n) \tag{14}$$

$$m(v,T_2) = a(v,T^n + \tfrac{1}{2}T_1) \tag{15}$$

$$m(v,T_3) = a(v,T^n + \tfrac{1}{2}T_2) \tag{16}$$

$$m(v,T_4) = a(v,T^n + T_3) \tag{17}$$

$$T^a = T^n + \tfrac{1}{6}T_1 + \tfrac{1}{3}T_2 + \tfrac{1}{3}T_3 + \tfrac{1}{6}T_4 \tag{18}$$

where T^n is the tracer concentration at timestep n and T^a is the tracer concentration after advection. Subsequently the diffusion term is solved using an implicit scheme:

$$m(v,T^{n+1}) + \tfrac{1}{2}d(v,T^{n+1}) = m(v,T^a) - \tfrac{1}{2}d(v,T^a) \tag{19}$$

giving the tracer concentration at the next timestep, T^{n+1}. This discretisation implies that no tracer enters or exits the domain through the boundaries. The solution variables are discretised using order-1 basis functions, and the problem is solved over a square domain with suitable initial conditions.

DOI: 10.1002/fld

5.1. Experimental setup

Our test hardware consists of three different GPUs: An NVidia GTX280, and NVidia GTX480, and an AMD Radeon 5870. The CPU used to test the baseline implementations was an Intel Xeon E5620 (Westmere EP) with 12GB of RAM.

CUDA and OpenCL implementations of the solver that implement both the Addto algorithm and the Local Matrix Approach have been produced for execution on the GPUs. The baseline CPU version is implemented within Fluidity [19], a finite element computational fluid dynamics code that has been chosen because it is a mature and optimised CPU implementation that is in production use for scientific applications, such as the Imperial College Ocean Model (ICOM) [20]. The Local Matrix Approach is not implemented in this version, as it is known to be less efficient than the Addto algorithm on CPUs for the low-order basis functions used in these experiments [9]. Execution is parallelised in Fluidity using domain decomposition, and different processes communicate using MPI. Node data structures are implemented using the element-wise storage layout in the CUDA implementation, and the node-wise layout is used in the CPU implementation.

The Intel v10.1 compilers with the `-O3` flag were used to compile the CPU code (v11.0 onwards cannot compile Fluidity at present due to compiler bugs), and the CUDA SDK 3.1 is used for the CUDA and OpenCL code running on NVidia GPUs. The AMD Stream SDK version 2.2 was used to compile the OpenCL for the AMD GPU, and to compile the OpenCL code to run on the CPU. The CUDA and OpenCL implementations use a *Conjugate Gradient* (CG) solver described in [21]; the baseline version makes use of the PETSc [22] CG solver. The simulation is run for 200 timesteps, with all computations using double precision arithmetic. Gmsh [23] was used to generate meshes varying in size between 58485 and 509260 elements. Using these mesh sizes allows the experiments to execute inside 1GB of RAM, which is the maximum available on the NVidia GTX280 and AMD 5870. Each simulation was run five times, and averages are reported.

6. RESULTS

Figure 10 shows the execution time for the OpenCL implementation with global assembly using colouring, and the local matrix approach. The largest mesh that could be used on this hardware consisted of 367321 elements, since the current driver implementation only allows 768MB of memory to be allocated. We note that the LMA implementation is consistently faster on this hardware, which is to be expected for a GPU implementation, as described in Section 4.1.

Figure 11 shows the execution time of the CUDA and OpenCL versions. We note that overall, the implementations of the LMA provide the best performance on this architecture. It can be seen that for some mesh sizes the OpenCL implementation outperforms the CUDA implementation and vice-versa for others. However, the difference in execution times on each mesh is negligible. Although both implementations of the Addto algorithm using colouring perform worse than the respective LMA version, the OpenCL implementation of this algorithm is noticeably slower than the CUDA implementation. We hypothesised in [12] that the LMA would outperform the Addto algorithm with colouring; these results validate this hypothesis. We note that using atomic operations instead of colouring leads to a significant slowdown.

Figure 12 shows the execution times of the benchmarks on the NVidia GTX480. We see that the performance of atomic operations is much increased on this architecture, and the execution times of the Addto algorithm with atomics and colouring are very similar. However, the LMA implementations are still more efficient than the Addto implementations. In the LMA implementations, there appears to be a constant overhead of using OpenCL compared to using CUDA. The OpenCL implementation of the Addto using colouring again is far slower than the OpenCL LMA implementation.

The gap in performance between the Addto and LMA in CUDA is much smaller in this architecture - it was suspected that the L1 cache that is present in the NVidia GTX480 may have reduced the overhead of the irregular accesses required by the Addto kernel. However, use of the NVidia Compute Profiler [24] showed that the L1 cache hit rate was less than 5% for this kernel.

DOI: 10.1002/fld

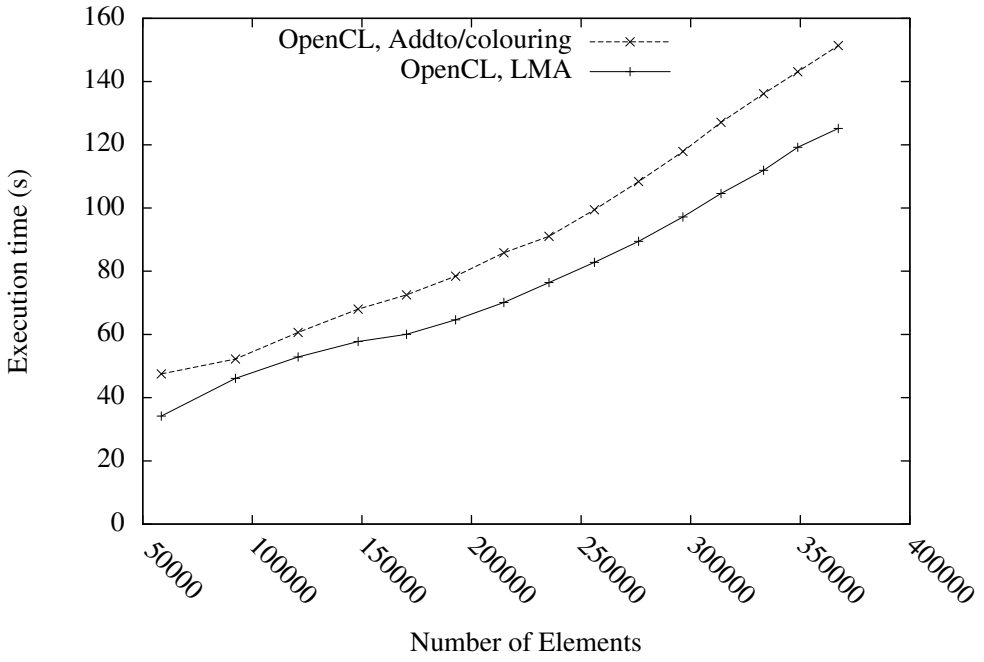

Figure 10. Execution times for simulations on the AMD Radeon 5870. Note that the Local Matrix Approach is more efficient on this hardware.

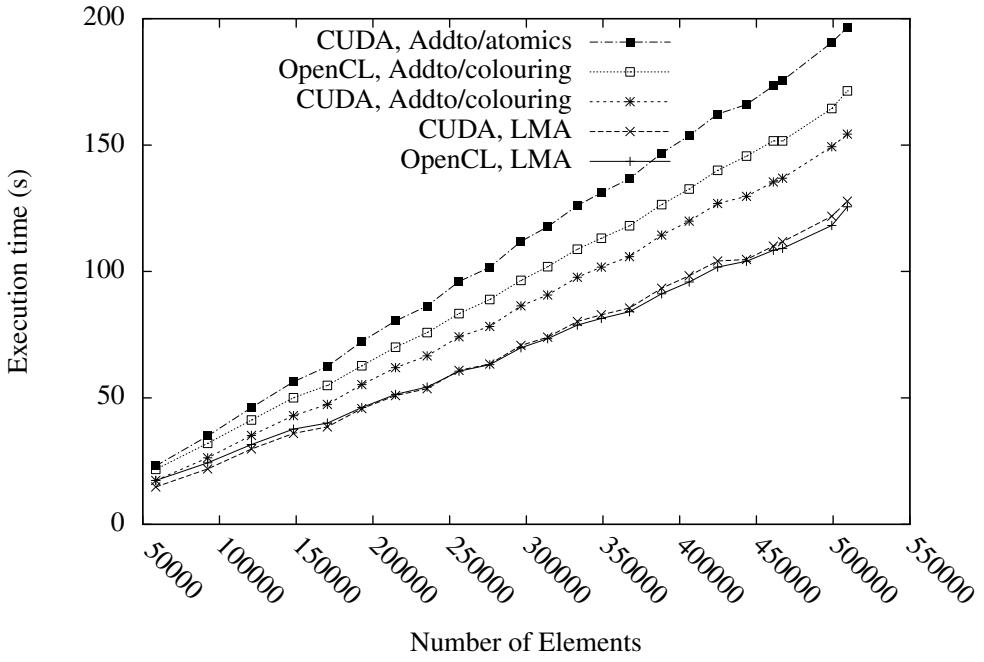

Figure 11. Execution times for simulations on the Nvidia GTX280. The CUDA and OpenCL implementations of the Local Matrix Approach are the fastest, and roughly equal in performance.

Figure 13 shows the execution time of the CPU baseline version on the 4-core Intel Xeon E5620. Because of the platform-portability of OpenCL, we were also able to compile and run the OpenCL code intended for the GPU on this platform. We see that the Addto algorithm is more efficient

DOI: 10.1002/fld

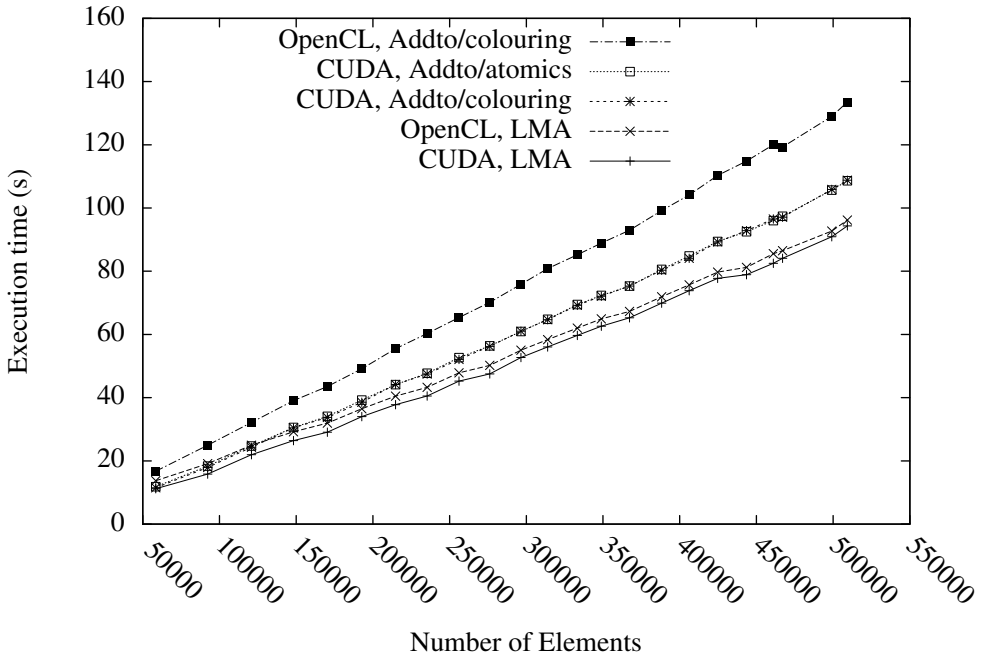

Figure 12. Execution times for the simulations on the Nvidia GTX480. Atomic operations appear far more efficient on this hardware than on the GTX280.

on the CPU. The OpenCL implementation of the Addto algorithm is slower than in the baseline version, since the expanded data layouts prevent the cache being exploited as there is no temporal locality, and makes less efficient use of memory bandwidth when coalescing is not required for high performance.

Figure 14 shows the relative speed of the fastest implementation on each architecture normalised against the size of the mesh. Note that this may not be taken directly as a measure of elements assembled per second, since the work required within the solve phase varies from mesh to mesh. We observe that the NVidia GTX480 provides a speedup of approximately an order of magnitude over the 4-core Intel Xeon E5620. There appears to be a large start-up cost when using the AMD 5870, which is not amortised until the mesh is on the order of 200000 elements. Additionally we note that its performance is approximately 50% of that of the NVidia 480GTX. This is a disappointing result given that the two architectures have similar peak memory bandwidth and double-precision arithmetic throughput. In the next section, we investigate the performance of the AMD card further.

6.1. AMD Radeon 5870 Performance Analysis

It was expected that the AMD card should have had similar performance to the NVidia GTX480 due to their similar theoretical peak double-precision arithmetic throughput and memory bandwidth [25, 26]. In order to investigate the performance on this hardware, we recorded the start and end execution times of each kernel by using OpenCL events and obtaining the profiling information for the event associated with each kernel invocation. Since recording the profiling information occupies some memory on the device, the largest mesh that this could be performed for was one with 148290 elements, and the simulation could only be run for 50 timesteps. This profiling information showed that the execution time of the kernels was relatively low compared to the entire execution time of the simulation. The sum of the kernel execution times obtained is compared with the total execution time as recorded by timing routines in the host code (which includes driver overhead) in Figure 15.

We see that the total execution time for all the kernels are very similar on both architectures. However, the total execution time on the AMD GPU is significantly higher, indicating that the driver

DOI: 10.1002/fld

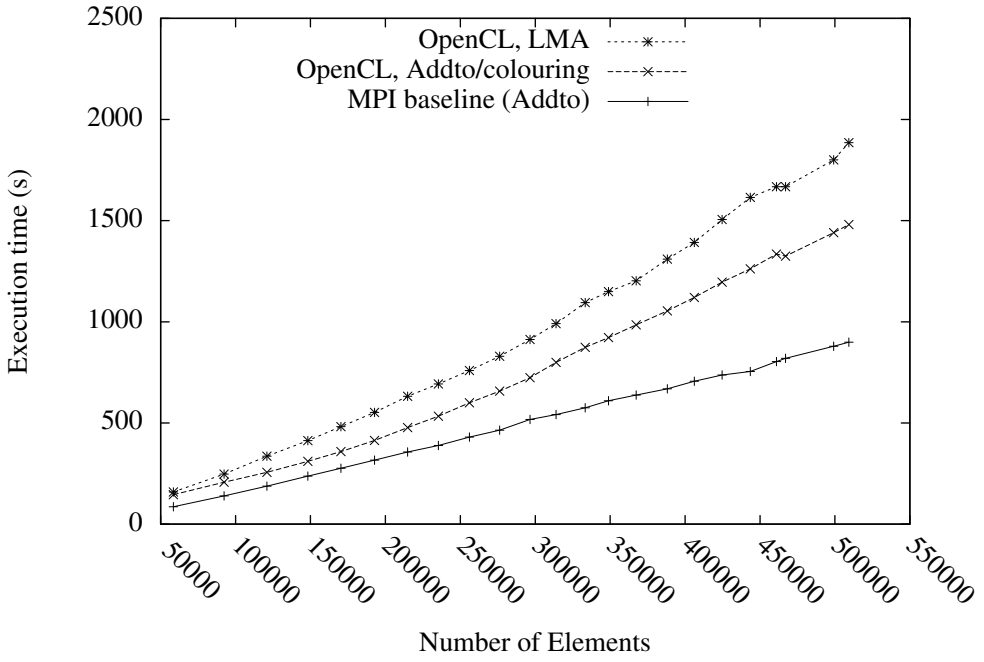

Figure 13. Execution times of the simulations on the 4-core Intel Xeon E5620. We see that the Addto algorithm is most efficient on this hardware, and the performance of the OpenCL code is less than the baseline due to its use of expanded data layouts.

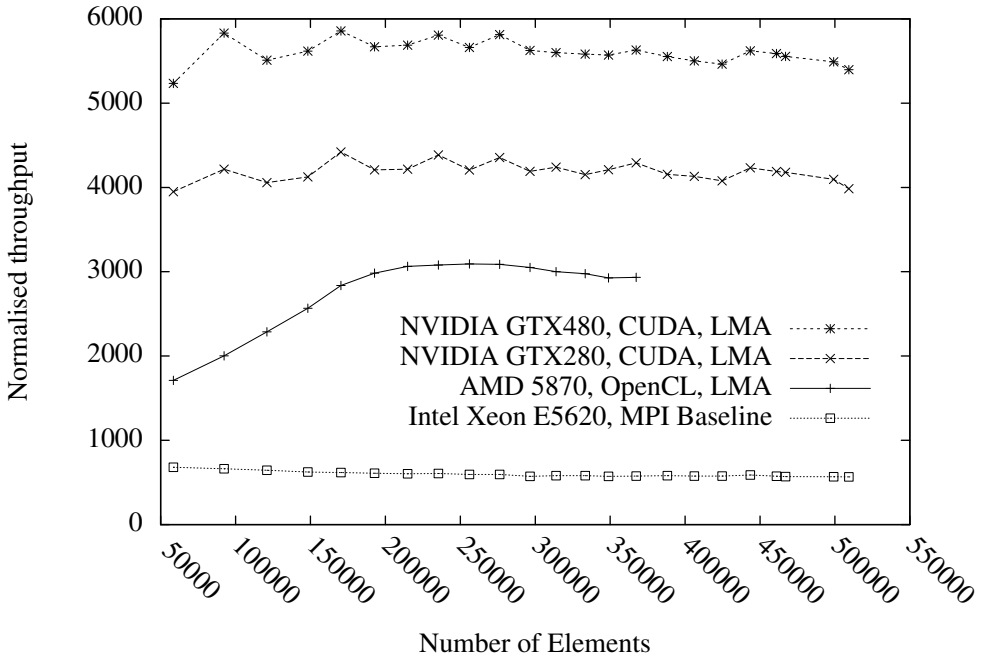

Figure 14. Speeds of the best implementations on each architecture normalised against mesh size. We see a performance gain of an order of magnitude between the NVidia GTX480 and CPU baseline implementation.

overhead of kernel launching is high. We conclude that the implementation of the AMD hardware is competitive with the NVidia hardware. However, the software implementation in its present form is

DOI: 10.1002/fld

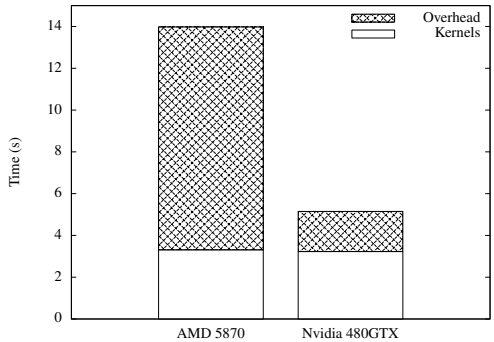

Figure 15. Kernel execution times and driver overheads. Kernel execution times are similar, but the driver overhead of the AMD card significantly lowers performance.

less competitive. Our experiments elsewhere have shown that the overhead of AMD's Linux driver appears to be significant overall and while we are unable to run this application under Windows, we would expect the results to be far closer in that setup.

6.2. Summary of Results

We conclude from the results from all architectures that the LMA is the fastest algorithm on GPU architectures. This is due to the increased coalescing and reduced control flow divergence afforded by this algorithm. On the CPU architectures, the Addto algorithm is the fastest approach, as a result of the cache, and overall lower memory bandwidth. The poor performance of the LMA in our OpenCL implementation compared to the Addto algorithm on the CPU agrees with the results of previous studies using other languages for their implementation [7, 8, 9]. We can conclude that the OpenCL implementation is performance-portable across similar architectures, but not across the gap between multi-core and many-core architectures.

7. RELATED WORK

There have been various investigations into the performance of various points in the implementation space of finite element methods. Most of those that we highlight involve implementations on GPU architectures.

An investigation into the implementation space for global assembly on GPUs is presented in [5]. The experimental investigation covered strategies mapping various granularities of computation to each thread. In particular, the investigations cover the use of one thread per non-zero in the global matrix, and using one thread per element. In the case of one thread per non-zero, the experiments were performed with the Addto operation fused with the computation of the local element matrix. The experiments are performed using a GPU implementation of a 2D heat equation solver run on previous generation GPUs, the NVidia 8800 and Tesla C1060. The general conclusions consist of two cases. Firstly, it is preferable to assemble using one thread per non-zero, holding intermediate results in shared memory when performing assembly for lower-order elements. Secondly, when the elements are higher-order, it is better to perform assembly using one thread per element, storing local matrices in global memory. We note that in the low-order case, there is less work to do per element for the local matrix assembly. In our implementation, we decided to use the one-thread-per-element strategy for global assembly since there is more work to do per-element in the local assembly for our experiments. Additionally we note that the experiments were performed using single-precision computations, as opposed to the double-precision computations used in our experiments.

In [2, 3], an implementation of a finite element code for seismic simulation is presented that is written in CUDA and uses MPI for communication between nodes. In this implementation, the

DOI: 10.1002/fld

Addto algorithm is used, but entire local matrices are coloured rather than rows of the local matrices, and the performance of assembly using atomic operations or the LMA is not investigated.

An implementation of the finite element method for hyperelastic material simulation is discussed in [1, 27], and optimisation of the code by fusing kernels is examined. We believe that this fusion of the kernels can also be represented at an abstract high level because individual terms (such as integrals) correspond to a kernel that evaluates them. However, this possibility of high-level representation is not discussed by the authors.

The performance of GPU kernels that evaluate local matrices for various polynomial orders of element is discussed in [4]. However, the other portions of the finite element method are not investigated.

In [7, 8, 9] the choice of optimal evaluation strategy is investigated, with consideration for the Addto algorithm, the LMA, and a tensor-based algorithm. It is shown that the optimal algorithm depends on factors including the dimension of the problem, polynomial order of the approximation, and the equation being discretised. The tradeoff to find the most efficient algorithm for solving an equation with a given tolerance is also discussed. However, these investigations are limited to CPU implementations.

7.1. Program Generation

Although some of these investigations consider similar dimensions in the implementation space to this one, we note that these investigations are performed at a low level, with the goal of discovering how to write efficient finite element codes, rather than how the high-level specifications relate to and allow the derivation of optimisations. We also draw attention to code generation tools that produce optimised implementations in similar domains:

The FEniCS Form Compiler [16, 28] generates code that implements methods described in UFL. The code generator uses optimisations based on algebraic manipulation of an intermediate (but high-level) representation in order to minimise the operation count of the generated code. Presently the code generator targets CPUs only.

OPlus2 [29, 30] is a framework for writing parallel programs that perform computations on unstructured meshes. It uses source-to-source translation to generate CUDA implementations of user-specified code. The abstraction provided by OPlus2 is a lower-level, more imperative one than that used to derive optimisations in this work such as the LMA.

8. CONCLUSIONS

We have investigated the implementation space for parallel global assembly in the finite element method using a variety of GPU platforms and a more traditional CPU architecture. These experiments, using an implementation of a finite element advection-diffusion solver.

Our results demonstrate that the algorithmic choice in finite element method implementations makes a big difference in performance, with the best choice depending on the target architecture. In particular, the Addto algorithm is preferable on CPU implementations, and the Local Matrix Approach is more efficient on GPU implementations. Additionally, the choice of data structures is also influenced by the target platform - data formats that contain redundant data are more efficient on many-core architectures, whereas structures that use indirection and do not store redundant data are more profitable on multi-core architectures.

We have demonstrated that although the OpenCL implementation is portable across CPU and GPU platforms, optimal performance cannot be achieved on all architectures without modifying the source code. This motivates the automation of code generation, allowing navigation of the various dimensions of the implementation space in order to pick the best implementation strategy for each context. This motivates our present work on a code-generation tool that transforms UFL sources into CUDA implementations. The code generator will be able to pick the choice of data structure and assembly algorithm depending on the context of the code generation. In order to select the

DOI: 10.1002/fld

optimal implementation, we believe it will be necessary to develop a performance model that assists in determining which algorithms will result in the greatest performance on the target hardware.

ACKNOWLEDGEMENTS

This work was supported by the NERC (DTG NE/G523512/1) and EPSRC (EP/E002412/1 and EP/I00677X/1). We express our thanks to AMD for donating the Radeon 5870 card that was used to perform some of the experiments. We would also like to thank Lee Howes of AMD for his input in tuning the OpenCL code.

REFERENCES

1. Filipovic J, Peterlik I, Fousek J. GPU Acceleration of Equations Assembly in Finite Elements Method - Preliminary Results. *SAAHPC : Symposium on Application Accelerators in HPC*, 2009.
2. Komatitsch D, Michéa D, Erlebacher G. Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA. *J. Parallel Distrib. Comput.* 2009; **69**(5):451–460.
3. Komatitsch D, Göddeke D, Erlebacher G, Michéa D. Modeling the propagation of elastic waves using spectral elements on a cluster of 192 GPUs. *Computer Science Research and Development* 2010; **25**(1-2):75–82, doi: 10.1007/s00450-010-0109-1.
4. Maciol P, Plaszewski P, Banas K. 3d finite element numerical integration on gpus. *Procedia Computer Science* 2010; **1**(1):1087 – 1094, doi:DOI:10.1016/j.procs.2010.04.121. URL http://www.sciencedirect.com/science/article/B9865-506HM1Y-48/2/d314d04e0723293b1abf9de45f265ed0, iCCS 2010.
5. Cecka C, Lew A, Darve E. Assembly of finite element methods on graphics processors. *International Journal for Numerical Methods in Engineering* 2011; **85**(5):640–669.
6. Aho AV, Lam MS, Sethi R, Ullman JD. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006. URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0321486811.
7. Cantwell C, Sherwin S, Kirby R, Kelly P. From h to p efficiently: Strategy selection for operator evaluation on hexahedral and tetrahedral elements. *Computers & Fluids* 2011; **43**:23–28, doi:DOI:10.1016/j.compfluid.2010.08.012. URL http://www.sciencedirect.com/science/article/B6V26-50V208D-1/2/5aa26c0ff5f8a0791ccdeb0651c17f61.
8. Cantwell CD, Sherwin SJ, Kirby RM, Kelly PHJ. From h to p efficiently: selecting the optimal spectral/hp discretisation in three dimensions. Math. Mod. Nat. Phenom. 2011. In press.
9. Vos PEJ, Sherwin SJ, Kirby RM. From h to p efficiently: Implementing finite and spectral/hp element methods to achieve optimal performance for low- and high-order discretisations. *J. Comput. Phys.* 2010; **229**(13):5161–5181, doi:http://dx.doi.org/10.1016/j.jcp.2010.03.031.
10. NVidia. NVidia CUDA Programming Guide Version 2.3.1. URL: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf August 2009.
11. Khronos Group T. *OpenCL 1.0 Working Specification* 2008. URL http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf.
12. Markall GR, Ham DA, Kelly PH. Towards generating optimised finite element solvers for gpus from high-level specifications. *Procedia Computer Science* 2010; **1**(1):1809 – 1817, doi:DOI:10.1016/j.procs.2010.04.203. URL http://www.sciencedirect.com/science/article/B9865-506HM1Y-76/2/3cb6d29fb608d89bc5460bd6d4a39b34, iCCS 2010.
13. Solin P, Segeth K, Dolezel I. *Higher-Order Finite Element Methods*. Chapman and Hall/CRC, 2003.
14. Karniadakis GEM, Sherwin SJ. *Spectral/hp Element Methods for CFD*. Oxford University Press, 1999.
15. Hesthaven JS, Warburton T. *Nodal Discontinuous Galerkin Methods*. Springer, 2007.
16. Logg A. Automating the finite element method. *Arch. Comput. Methods Eng.* 2007; **14**(2):93–138.
17. NVidia. Fermi white paper. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf, Retrieved 19 May 2011. 2010.
18. Barrett R, Berry M, Chan TF, Demmel J, Donato J, Dongarra J, Eijkhout V, Pozo R, Romine C, der Vorst HV. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM: Philadelphia, PA, 1994.
19. Gorman G, Piggott M, Farrell P. About Fluidity. URL: http://amcg.ese.ic.ac.uk/index.php?title=FLUIDITY November 2009.
20. Piggott MD. About ICOM. http://amcg.ese.ic.ac.uk/index.php?title=ICOM, Retrieved 30 Sep 2010.
21. Markall G, Kelly PHJ. Accelerating Unstructured Mesh Computational Fluid Dynamics Using the NVidia Tesla GPU Architecture. *ISO Report*, Imperial College London 2009.
22. Balay S, Buschelman K, Gropp WD, Kaushik D, Knepley MG, McInnes LC, Smith BF, Zhang H. PETSc Web page 2009. Http://www.mcs.anl.gov/petsc.
23. Geuzaine C, Remacle JF. Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities. *Int. J. Numerical Methods in Engineering* 2009; **79**(11):1309–1331.
24. NVidia. *Visual Profiler User Guide*.

25. AMD. AMD Radeon 5870 Specifications. http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/hd-5870/Pages/ati-radeon-hd-5870-overview.aspx#2, Retrieved 30 Sep 2010.
26. NVidia. NVidia Geforce GTX480 Specifications. http://www.nvidia.com/object/product_geforce_gtx_480_us.html, Retrieved 30 Sep 2010.
27. Filipovic J, Fousek J, Matyska L, Peterlik I. Decomposition-Fusion Scheme for Medium-Grained Problems on GPU. Unpublished manuscript 2010.
28. Kirby RC, Logg A. A compiler for variational forms. *ACM Transactions on Mathematical Software* 2006; **32**(3):417–444. URL http://home.simula.no/~logg/pub/papers/KirbyLogg2006a.pdf.
29. Sharif Z. An AEcute framework for accelerating unstructured mesh based computation using the CUDA programming model. Master's Thesis, Imperial College London 2010.
30. Giles M. A framework for parallel unstructured grid applications on GPUs. Plenary talk at the SIAM conference on parallel processing for scientific computing (PP10), Seattle. February 2010.

DOI: 10.1002/fld