Towards generating optimised finite element solvers for GPUs from high-level specifications

Graham R. Markall[*], David A. Ham[†], Paul H. J. Kelly[*]

Abstract

We argue that producing maintainable high-performance implementations of finite element methods for multiple targets requires that they are written using a high-level domain-specific language. We make the case for using one such language, the *Unified Form Language* (UFL), by discussing how it allows the generation of high-performance code from maintainable sources. We support this case by showing that optimal implementations of a finite element solver written for a Graphics Processing Unit and a multicore CPU require the use of different algorithms and data formats that are embodied by the UFL representation. Finally we describe a prototype compiler that generates low-level code from high-level specifications, and outline how the high-level UFL representation can be lowered to facilitate optimisation using existing techniques prior to code generation.

1 Introduction

The development of finite element codes in low-level languages is complicated and error prone. When a code is ported to a new architecture, much of it must be rewritten in a new language, requiring a large amount of time and effort. This process is complicated by the fact that optimal data layouts and access patterns differ between targets, especially when execution of the code spans multiple architectures. Additionally, the optimal choice of algorithm that implements a given operation depends on characteristics of the target hardware, and even the parameters of a specific problem. To produce efficient implementations in a low-level language, developers must maintain multiple algorithm implementations for multiple targets.

The FEniCS project [9] has shown that the *Unified Form Language* (UFL) provides an appropriate level of abstraction of the finite element method for generating optimised code from maintainable sources. Using UFL for writing finite element codes is desirable as it eliminates many time-consuming and error-prone tasks required when developing in a low-level language, and prevents many common errors. In addition to this, we propose that unmodified UFL sources can be translated into low-level code for multiple hardware platforms with target-specific optimisations.

The contributions of this paper are: a high-performance implementation of a finite element solver for an advection-diffusion problem written using NVidia CUDA (Section 3), a prototype implementation of a compiler that generates CUDA code from UFL sources (Section 4.2), and a discussion of how the UFL representation can be lowered to allow the use of loop nest optimisations (Section 4.3). We begin with a brief overview of the CUDA programming language (Section 1.1) and a description of the main stages of the finite element method (Section 2) before discussing options for its implementation.

[*]Department of Computing, Imperial College London

[†]Department of Earth Science & Engineering and Grantham Institute for Climate Change, Imperial College London

1.1 CUDA

CUDA [12] is a language for programming NVidia's Tesla *Graphics Processing Units* (GPUs). GPUs have a large memory bandwidth and many processing cores, making them ideal for computational science applications. Execution on a GPU is performed by launching individual *kernels* that are executed by many threads in parallel. 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. The workload must be decomposed into many (thousands) of data-parallel tasks that can be mapped to individual threads to fully utilise the GPU hardware.

We highlight two performance considerations. First, groups of 32 threads (referred to as *warps*) all share a single program counter, and execute the same code path concurrently. When threads within a warp take different paths, execution is serialised between these two paths, reducing performance. Second, *coalesced* memory access is needed for high memory bandwidth utilisation, and is achieved when groups of 16 threads concurrently access data within a 128-byte aligned memory window. For further details of the Tesla architecture and CUDA programming language, see [12].

2 The Finite Element Method

The finite element method is used for discretising the weak form of partial differential equations. 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 i in the domain, an $N_e \times N_e$ matrix, $\mathbf{M}_i^e$, and an N_e-length vector, $\mathbf{b}_i^e$, are computed, where N_e 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 elements using Gaussian quadrature. In most implementations, every element has the same number of nodes.

Global Assembly. The local matrices, $\mathbf{M}_i^e$, and vectors, $\mathbf{b}_i^e$, are used to form a *global matrix*, $\mathbf{M}$, and *global vector*, $\mathbf{b}$, respectively. This process couples the contributions of elements together.

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) $\mathbf{y} = \mathbf{Mv}$.

For a complete treatment of the method, see [7]. We shall examine the global assembly phase, which consists of performing the following computations:

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

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

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 i-th block is $\mathbf{M}_i^e$, and $\mathbf{b}^e$ is a vector of stacked $\mathbf{b}_i^e$. We shall examine algorithms that can be used to implement these computations, as the optimal choice of algorithm depends on the target hardware.

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

Consider a two-element, three-node decomposition of a 1-dimensional domain (see Figure 1). In this example, the matrices and vector are:

$$\mathcal{A} = \begin{bmatrix} 1 & & \\ & 1 & \\ & 1 & \\ & & 1 \end{bmatrix} \qquad \mathbf{M}^e = \begin{bmatrix} m^1_{11} & m^1_{12} & & \\ m^1_{21} & m^1_{22} & & \\ & & m^2_{11} & m^2_{12} \\ & & m^2_{21} & m^2_{22} \end{bmatrix} \qquad \mathbf{b}^e = \begin{bmatrix} b^1_1 \\ b^1_2 \\ b^2_1 \\ b^2_2 \end{bmatrix}$$

where m^e_{ij} is the i,j-th term of the local matrix for element Ω^e, and b^e_i is the i-th term of the local vector for element Ω^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:

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

Algorithms 1 and 2 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.

Algorithm 1: Addto for global matrix assembly.

```
M = 0 ;
foreach Element e do
    for i ← 1 to N_e do
        for j ← 1 to N_e do
            M[map[e][i], map[e][j]] += M^e[i, j] ;
```

Algorithm 2: Addto for global vector assembly.

```
b = 0 ;
foreach Element e do
    for i ← 1 to N_e do
        b[map[e][i]] += b^e[i] ;
```

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 must be used to ensure correctness. Second, $\mathbf{M}$ is often stored using a format such as

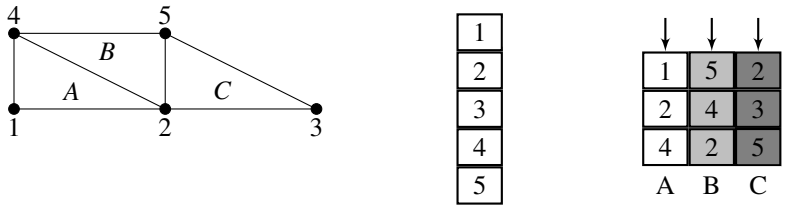


Figure 2: *Left:* A 2D domain decomposed into three elements. *Middle:* Node data layout in CPU implementation. *Right:* Node data layout in GPU implementation. Threads accessing data in different elements (arrowed) achieve coalescing.

compressed sparse row (CSR). 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.

To avoid these issues, we can derive an alternative algorithm, referred to as the *Local Matrix Approach* (LMA), 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:

$$\mathbf{y} = \left(\mathcal{A}^T \left(\mathbf{M}^e \left(\mathcal{A}\mathbf{v}\right)\right)\right) \tag{3}$$

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. We demonstrate in Section 3 that the optimal choice of algorithm depends on the target hardware.

2.1 Data Format Considerations

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 2).

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.

Maintaining low-level implementations for multiple devices that use different data structures is time-consuming and error-prone. The complexity of managing multiple data structures is exacerbated if data must be marshalled between devices, for example in the case when computation is distributed between a CPU and a GPU. A UFL compiler that is aware of the data formats of its target devices can automatically generate code that marshals the required data based on a UFL description of a method, as it is possible to determine the items of data that are required from the UFL source.

3 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:

$$\frac{\partial T}{\partial t} + \mathbf{u}\nabla T = \nabla \cdot \bar{\bar{\mu}} \cdot \nabla T$$

where T is the concentration of a tracer, t is time, $\mathbf{u}$ is velocity, and $\bar{\bar{\mu}}$ 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. The system is discretised using order-1 basis functions. A split scheme is used, solving for advection first and then diffusion at each time step. The advection term is timestepped using a 4th-order Runge-Kutta scheme, and the diffusion term is timestepped using an implicit theta scheme. The problem is solved over a square domain with suitable initial conditions. For further details of the test implementation, see [10]. We compare with a CPU implementation to demonstrate that the optimal choice of algorithm depends on the target hardware.

3.1 CUDA and CPU Implementations and Experimental Setup

CUDA Implementations of the solver that implement both the Addto algorithm and the Local Matrix Approach have been produced. The Local Matrix Approach is implemented by considering the computation in Equation 3 in three stages:

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

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.

The baseline version is implemented within Fluidity [5], a finite element computational fluid dynamics code that has been chosen because it is a mature and optimised CPU implementation. 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 low-order basis functions [15]. 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 test hardware consists of an Intel Core 2 Duo E8400, 2GiB RAM, and an NVidia 280GTX GPU. The Intel v10.1 compilers with the `-O3` flag were used for the CPU code (v11.0 onwards cannot compile Fluidity due to compiler bugs), and the CUDA SDK 2.2 is used for CUDA code. The CUDA implementation uses a *Conjugate Gradient* (CG) solver described in [11]; the baseline version make use of the PETSc [2] CG solver. The simulation is run for 200 timesteps, with all computations using double precision arithmetic. Gmsh [4] was used to generate meshes varying in size between 28710 and 331714 elements. Each simulation was run five times, and averages are reported.

3.2 Results

Figure 3 shows the total time taken by each CUDA implementation to run the entire simulation, and Figure 4 shows their speedup relative to the baseline version running on 2 cores. We see that the LMA implementation is up to 2.2 times faster than the Addto implementation on the GPU, and is over an order of magnitude faster than the baseline implementation.

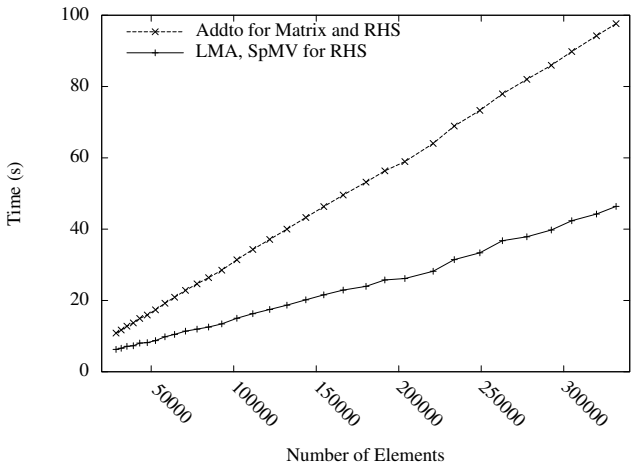


Figure 3: Total execution time of GPU implementations.

Figure 5 shows the total time taken for the local and global assembly phases in the CUDA implementations. We observe that the Local Matrix Approach is faster than the Matrix Addto algorithm, and that it is faster to assemble the global vector by computing the product $\mathcal{A}^T\mathbf{b}^e$. Table 1 shows the total time spent inside the SpMV kernels for each implementation for the largest and smallest mesh sizes. The cost of computing $\left(\mathcal{A}^T\left(\mathbf{M}^e\left(\mathcal{A}\mathbf{v}\right)\right)\right)$ (in the LMA implementation) is up to 2.5 times that of computing $\mathbf{Mv}$ (in the Addto implementation). The performance increase of the LMA implementation is a tradeoff between the decrease in the assembly time, and the increase in the SpMV computation time.

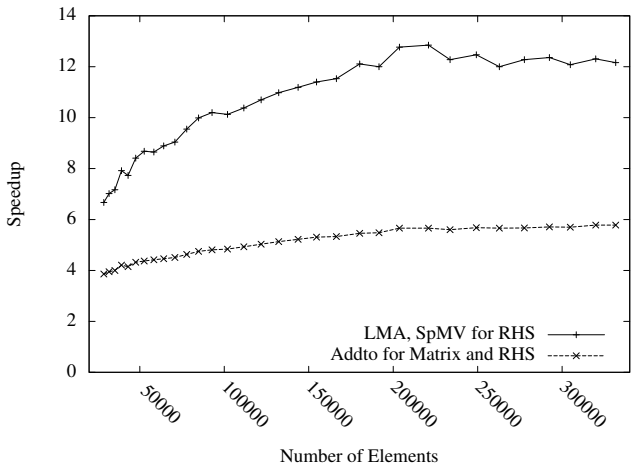


Figure 4: Speedup of GPU implementations relative to the baseline executing using 2 cores.

Elements	$\left(\mathcal{A}^T\left(\mathbf{M}^e\left(\mathcal{A}\mathbf{v}\right)\right)\right)$	$\mathbf{Mv}$
28710	1.91×10^6	8.48×10^5
331714	2.45×10^7	9.85×10^6

Table 1: Total time spent computing each product on the largest and smallest meshes (in μsec), recorded using the CUDA Profiler.

We also investigated using graph colouring to eliminate atomic operations in the Addto implementation (as used in [8]). We replaced atomic operations with equivalent non-atomic operations. The resulting implementation produced incorrect results, but gave an upper bound on the performance increase facilitated by colouring. The assembly phase ran 25% faster with non-atomic operations, corresponding to a 10% speedup in the entire simulation. Since the performance of the LMA implementation is far greater than this, it is unnecessary to produce an implementation that uses colouring.

3.3 Remarks

Our results show that the optimal algorithm depends on the target architecture. We speculate that it is also problem-dependent. In a 2D domain the average variance of nodes is approximately 6. In a 3D domain, the variance is around 24, and the overheads of storage and computation for the local matrix approach are four times greater than in 2D. This extra overhead may decrease performance to the point where it is more efficient to use the Addto algorithm. Our further work involves investigating the performance in this case.

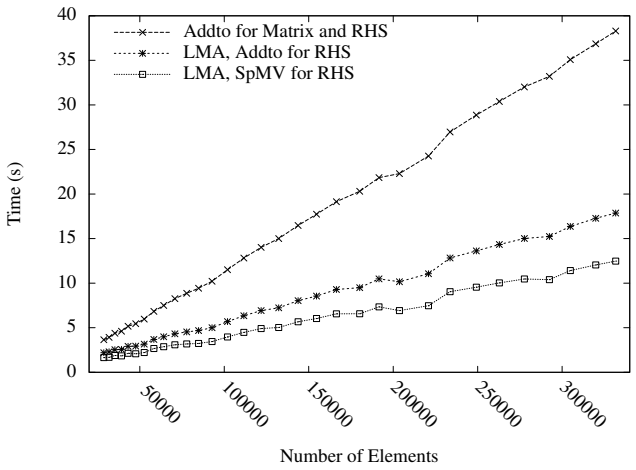


Figure 5: Execution time of the assembly phases for each CUDA implementation.

4 Compiling the Unified Form Language

4.1 Poisson's Equation - A Motivating Example

Here we discuss how UFL can be used to describe the assembly and solution of a partial differential equation. For a complete reference of UFL, see [1]. Poisson's Equation, and a weak form are:

$$\nabla^2 u = f \tag{4}$$

$$\int_\Omega \nabla v \cdot \nabla u \, \mathrm{d}X = \int_\Omega v f \, \mathrm{d}X \tag{5}$$

We assume the boundary condition $\int_{\partial\Omega} v \nabla u \cdot \mathbf{n} \, \mathrm{d}s = 0$ to simplify the example. Figure 6 gives the UFL code for the assembly and solution of the weak form. Lines 1 and 2 instruct the compiler that v and u are test and trial functions, which is known from the mathematical formulation of the problem. The known function f is specified as a function of coordinates within the domain. `A` and `RHS` specify the left- and right-hand sides of Equation 5. The final line specifies that these forms are to be assembled into a linear system, which is solved to find the solution to the problem. The `solve` keyword is an addition to UFL, which is usually provided by tools that are part of the FEniCS project.

Note that Figure 6 is a complete specification of how the problem may be solved using the finite element method, yet contains no implementation-specific information. This provides the flexibility to generate code for multiple architectures, using alternative algorithms - the code is effectively "future-proofed". In particular, a low-level implementation using either the Addto algorithm or the Local Matrix Approach may be generated from this source. Compare this with a direct implementation in a low-level language, for which it is difficult to transform the data layout or implementation algorithm. The UFL compiler eventually commits to specific aspects of the low-level implementation during one or more of its passes.

```
v=TestFunction(P)
u=TrialFunction(P)
f=Function(P, sin(x[0])+cos(x[1]))
A=dot(grad(v),grad(u))*dx
RHS=v*f*dx
P = solve(A,RHS)
```

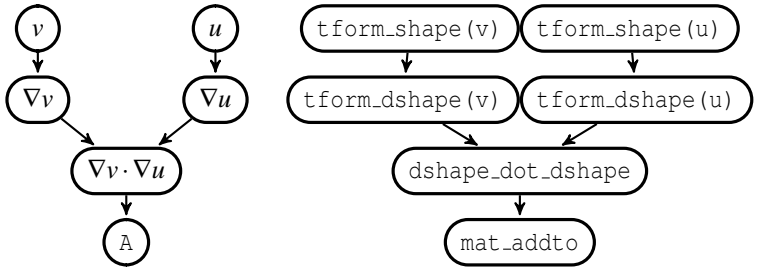
Figure 6: UFL code for the assembly and solution of Poisson's Equation. `P` is assumed to be some finite element space.

Targeting a new platform is accomplished by the development of a new compiler backend, without modifying the UFL sources. This allows the concerns of different developers to be separated: the work of mathematicians who test and implement new schemes is decoupled from the work of those whose concern is the low-level implementation of codes. This separation eases the development of finite element codes by eliminating a large proportion of the repetitive and error-prone tasks that are usually required. The generated code can be better optimised than handwritten code, as optimisations often cut across the boundaries of the abstractions required for developing software in low-level languages.

4.2 A Prototype Compiler

A first step in experimenting with generating CUDA code from UFL involved the implementation of a compiler that performs a syntax-directed translation of the UFL code to CUDA code. The generated code uses a library of kernels that perform common operations in finite element assembly. A subset of the kernels in this library are shown in Table 2. These kernels perform quadrature-based assembly, rather than the tensor-based method that is used in the FEniCS project.

The compiler inputs UFL using the FEniCS UFL distribution [1] to produce *Directed Acyclic Graphs* (DAGs) of the operations specified in a UFL source. Each DAG node is converted to a call to a kernel implementing the required operation. This DAG of kernel calls is passed to a code generator that is implemented using the ROSE Compiler Infrastructure [14]. Examples of the DAGs for the left-hand side of Equation 5 are shown in Figure 7.



(a) Expression DAG. (b) Kernel DAG.

Figure 7: DAGs for the form $\int_\Omega \nabla v \cdot \nabla u \, \mathrm{d}X$.

Kernel	Operation
`tform_shape`	Transforms the shape functions of a reference element into physical space.
`tform_dshape`	Transforms the derivatives of the shape functions of a reference element into physical space.
`dshape_dot_dshape`	Computes $\int_\Omega \nabla v \cdot \nabla u \, \mathrm{d}X$.
`shape_shape`	Computes $\int_\Omega vu \, \mathrm{d}X$.
`mat_addto`	Adds local matrices into a global matrix using the Addto algorithm.

Table 2: A subset of kernels in the CUDA kernel library.

The CUDA code generated by this compiler for the Poisson problem produces identical results to a handwritten CUDA implementation, as well as a CPU implementation of the same problem. Although the generated code executes faster than the CPU implementation for large meshes, we do not investigate its performance as there is only a limited speed improvement that can be gained for steady-state problems.

4.3 Further UFL Compiler Development

Although the prototype compiler demonstrates the feasibility of generating CUDA code from UFL sources, the requirement for a library of handwritten CUDA kernels limits its output to a pre-defined set of forms optimised by hand. Here we describe an intermediate phase that lowers the UFL representation to one amenable to optimisation with established techniques before being used to generate CUDA kernels. Consider one term from the weak form of the Helmholtz equation [7]:

$$\int_\Omega \nabla v \cdot \nabla u + \lambda vu \, \mathrm{d}X \tag{6}$$

There are several combinations of kernels that implement the local assembly phase of this term that can be implemented. We can enumerate these possibilities by building an *Intermediate Representation* (IR) that provides a high-level semantic representation of each term. Sub-terms of the intermediate representation are determined by working bottom-up from leaf nodes to identify the smallest set of nodes that describes the assembly of a local matrix. Higher sub-terms are identified as the addition or scalar multiplication of lower-sub terms.

The IR for Equation 6 is shown in Figure 8. As there are four sub-terms, up to four separate kernels may perform local assembly for this term. Using more kernels increases the memory bandwidth requirements; however, larger kernels require more resources, decreasing the total parallelism [3]. Instead of building a performance model to evaluate each candidate implementation, a pragmatic approach is to lower this representation to one that can be optimised using existing techniques.

Each of the nodes at the root of the sub-terms may be lowered to a loop over a certain index. For example, the `sum` node corresponds to a loop over the elements that sums the local matrices produced by the lower sub-terms. The generated loop nest can be optimised using standard techniques, for example in the polyhedral model [13] or the Æcute framework [6]. Our current work involves implementing a compiler that performs this lowering.

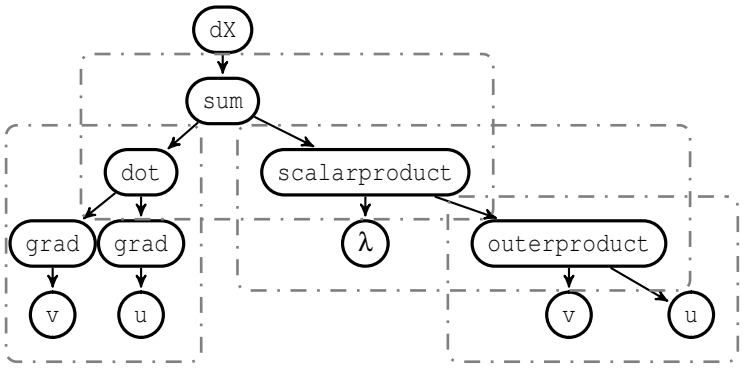


Figure 8: Intermediate Representation of the term $\int_\Omega \nabla v \cdot \nabla u + \lambda v u \, \mathrm{d}X$. Sub-terms are indicated by dotted outlines.

5 Conclusions and future work

UFL is a desirable language in which to write finite element codes due to its closeness to the mathematical representation of the finite element method. We have shown that the optimal choice of algorithm and data format used by a low-level implementation depends on the target hardware. The UFL representation provides freedom of choice in the algorithm and data format used by generated code and therefore provides the right level of abstraction for generating optimised code for multiple targets. Furthermore, the UFL representation may be lowered before code generation to facilitate optimisation using existing techniques.

Our long-term project is to rewrite a large portion of Fluidity using UFL. This process will take place gradually, as integration of UFL codes within its Fortran sources is made possible by the use of a UFL compiler whose output makes use of data structures in Fluidity. The result of this work will be a portable and maintainable high-performance code that allows aggressive exploitation of future architectures.

Acknowledgements

The authors wish to thank Alexander Monakov for his suggestion to implement stages 1 and 2 of the LMA in a single kernel, and Anton Lokhmotov for his helpful comments and feedback. This work was partially supported by the NERC (DTG NE/G523512/1) and EPSRC EP/E002412/1.

References

[1] Martin Alnæs and Anders Logg. UFL Specification and User Manual 0.1. URL: `http://www.fenics.org/pub/documents/ufl/ufl-user-manual/ufl-user-manual.pdf`, April 2009.

[2] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc Web page, 2009. http://www.mcs.anl.gov/petsc.

[3] Jiri Filipovic, Igor Peterlik, and Jan Fousek. GPU Acceleration of Equations Assembly in Finite Elements Method - Preliminary Results. In *SAAHPC : Symposium on Application Accelerators in HPC*, July 2009.

[4] Christophe Geuzaine and Jean-François Remacle. Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities. *Int. J. Numerical Methods in Engineering*, 79(11):1309–1331, 2009.

[5] Gerard Gorman, Matthew Piggott, and Patrick Farrell. About Fluidity. URL: `http://amcg.ese.ic.ac.uk/index.php?title=FLUIDITY`, November 2009.

[6] Lee W. Howes, Anton Lokhmotov, Alastair F. Donaldson, and Paul H.J. Kelly. Deriving efficient data movement from decoupled access/execute specifications. In *Proceedings of the 4th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC)*, volume 5409 of *Lecture Notes in Computer Science*, pages 168–182. Springer, 2009.

[7] George E. M. Karniadakis and Spencer J. Sherwin. *Spectral/hp Element Methods for CFD*. Oxford University Press, 1999.

[8] Dimitri Komatitsch, David Michéa, and Gordon Erlebacher. Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA. *J. Parallel Distrib. Comput.*, 69(5):451–460, 2009.

[9] A. Logg. Automating the finite element method. *Arch. Comput. Methods Eng.*, 14(2):93–138, 2007.

[10] Graham Markall. Generatively Programming Galerkin Projections on General Purpose Graphics Processing Units. Master's thesis, Imperial College London, 2009. `http://www.doc.ic.ac.uk/~grm08/graham_markall_msc_report.pdf`.

[11] Graham Markall and Paul H. J. Kelly. Accelerating Unstructured Mesh Computational Fluid Dynamics Using the NVidia Tesla GPU Architecture. ISO Report, Imperial College London, 2009.

[12] 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.

[13] Louis-Noel Pouchet, Cedric Bastoul, Albert Cohen, and Nicolas Vasilache. Iterative optimization in the polyhedral model: Part i, one-dimensional time. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 144–156, Washington, DC, USA, 2007. IEEE Computer Society.

[14] Dan Quinlan, Chunhua Liao, Thomas Panas, Robb Matzke, Markus Schordan, Rich Vuduc, and Qing Yi. ROSE: A Tool For Building Source-to-Source Translators. User Manual (version 0.9.4a). `http://www.rosecompiler.org/ROSE_UserManual/ROSE-UserManual.pdf`, Retrieved 15 September 2009, 2009.

[15] Peter E. J Vos, Spencer J. Sherwin, and Robert M. Kirby. From h to p efficiently: implementing finite and spectral/hp element discretisations to achieve optimal performance at low and high order approximations. Submitted to Journal of Computational Physics. `http://www2.imperial.ac.uk/ssherw/spectralhp/papers/JCP-VoShKi-09.pdf`, October 2009.