IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Generatively Programming Galerkin Projections on General Purpose Graphics Processing Units

By

Graham Markall

Supervisor: Prof. Paul Kelly
Second Marker: Dr. Tony Field

Submitted in partial fulfilment of the requirements for the MSc Degree in Advanced Computing of Imperial College London

September 2009

Abstract

This report presents the results of a preliminary investigation into using abstract specifications of finite element methods to generate code that performs the assembly of a system of linear equations on multicore architectures, with a focus on NVidia's CUDA language. This investigation has been conducted with the goal of integrating generated code into Fluidity, a general-purpose computational fluid dynamics code. We survey and evaluate CUDA implementations of finite element assembly, in particular examining the optimisations necessary for high performance, and examine the state of the art in the automatic generation of finite element assembly code.

CUDA implementations of the assembly phase of Fluidity test programs that solve Poisson's Equation and an Advection-Diffusion equation are presented. We demonstrate a performance improvement of almost an order of magnitude over a multicore CPU implementation for the Advection-Diffusion equation on typical hardware performing computations using double-precision arithmetic. We identify that the performance of the CUDA implementation is limited by the use of atomic operations, and the use of the Compressed Sparse Row matrix format, both of which are costly. We outline how further significant performance gains may be achieved by modifying our implementation to overcome these limitations.

These implementations are used to guide the design of a prototype compiler, which we use to demonstrate the feasibility of generating CUDA code from abstract specifications. We outline further work based on the research we have conducted, with a long-term goal of converting the low-level Fortran implementations of finite element assembly in Fluidity into high-level abstract specifications, to facilitate the exploitation of future multicore architectures.

Acknowledgements

People who I would like to thank for their contributions to the success of this project are:

- Paul Kelly, who has dedicated a large number of hours to the supervision of this project. His guidance, support, and helpful suggestions throughout the duration of this project have been invaluable. I would also like to express gratitude for the support he gave me throughout the first ISO, and general advice throughout the duration of the MAC course.
- David Ham, for many things, including: spending a large amount of time explaining the finite element method and the Fluidity codebase; for writing the test problem without which I would not have been able to demonstrate success in this project; for making time to answer all my questions and read drafts of my report; for his guidance and encouragement; and for making this project a possibility.
- Anton Lokhmotov, for spending a great deal of time examining drafts of my report and providing lots of helpful suggestions and comments.
- Francis Russell, for many interesting discussions, and for prompting me to think about how iteration might be captured in UFL.
- Lee Howes, Gerard Gorman and Patrick Farrell for their advice and suggestions regarding the performance results, and for interesting discussions.

Contents

1 Introduction **1**
- 1.1 Project Outline . . . 1
- 1.2 Contributions . . . 2
- 1.3 Presentation . . . 2

2 Background **3**
- 2.1 Introduction . . . 3
- 2.2 The NVidia Tesla GPU Architecture and CUDA Programming Language . . . 3
  - 2.2.1 The Parallel Programming Model . . . 5
  - 2.2.2 The Memory Hierarchy . . . 5
  - 2.2.3 Introducing CUDA . . . 7
  - 2.2.4 Remarks . . . 7
  - 2.2.5 Other Multicore Architectures and Languages . . . 8
- 2.3 The Finite Element Method . . . 8
  - 2.3.1 Discretising the Domain . . . 9
  - 2.3.2 Assembly and Solution . . . 9
  - 2.3.3 Boundary Conditions . . . 10
- 2.4 Fluidity . . . 11
  - 2.4.1 `test_laplacian` . . . 11
  - 2.4.2 `test_advection_diffusion` . . . 12
- 2.5 The Unified Form Language . . . 13
  - 2.5.1 UFL Compiler Optimisations . . . 14
- 2.6 Summary . . . 14

3 Related Work **17**
- 3.1 Introduction . . . 17
- 3.2 Finite Element Assembly on GPUs . . . 17
  - 3.2.1 The Genesis of the Finite Element Method on Graphics Processors . . . 17
  - 3.2.2 Hyperelastic Material Simulation . . . 17
  - 3.2.3 Discontinuous Galerkin on GPUs . . . 19
  - 3.2.4 Soft Tissue Modelling in the SOFA Framework . . . 21
  - 3.2.5 High-Order Earthquake Modelling . . . 23
  - 3.2.6 Finite Element in CPU/GPU Clusters . . . 23
- 3.3 Generating Execution Schedules for Tensor Contractions . . . 23
- 3.4 PyCUDA . . . 24
- 3.5 Generative Programming/Automation of Finite Element Methods . . . 25
  - 3.5.1 Remarks . . . 25
- 3.6 Conclusions . . . 26

4 Implementation of Finite Element Assembly using CUDA **29**
- 4.1 Introduction . . . 29
- 4.2 Initial Implementation of the assembly routine of `test_laplacian` using CUDA . . . 29
  - 4.2.1 The Assembly Loop in Fortran . . . 29

4.2.2 Implementation of Boundary Conditions . . . 31
4.2.3 Translation Methodology . . . 31
4.2.4 Integration with a GPU Conjugate Gradient Solver . . . 33
4.2.5 Testing . . . 33
4.2.6 Initial Performance Results . . . 34
4.2.7 Optimising Kernels . . . 34
4.2.8 Ensuring Coalesced Memory Accesses . . . 36
4.2.9 Post-Optimisation Performance . . . 37
4.3 Implementation of the Assembly Phase of `test_advection_diffusion` Using CUDA 37
4.3.1 Kernels Used in this Implementation . . . 39
4.3.2 Testing . . . 40
4.4 Performance Results and Analysis . . . 43
4.4.1 Performance of the Assembly Phase . . . 43
4.4.2 Speedup and Throughput . . . 43
4.4.3 Overall GPU Performance . . . 45
4.4.4 Performance Improvement . . . 49
4.5 Conclusions . . . 52

5 A UFL Compiler for CUDA 53
5.1 Introduction . . . 53
5.2 Design . . . 53
5.2.1 The Backend . . . 53
5.2.2 A UFL Frontend . . . 54
5.2.3 Integration With Fluidity . . . 57
5.3 Implementation . . . 57
5.3.1 The Python Frontend . . . 57
5.3.2 The Code Generation Backend . . . 58
5.4 Testing . . . 59
5.4.1 Generation of Test Input for the Backend . . . 59
5.4.2 Generation of Test Input for the Frontend . . . 60
5.4.3 Generation of Further Inputs . . . 60
5.5 Conclusion . . . 61

6 Evaluation 63
6.1 Introduction . . . 63
6.2 Examination of the Implementations of the Assembly Phase . . . 63
6.3 Discussion of the UFL Compiler . . . 64
6.4 Examination of UFL . . . 65
6.5 Summary . . . 66

7 Conclusions and Future Work 67
7.1 Conclusions . . . 67
7.2 Further Work . . . 67
7.2.1 Performance Optimisation of the GPU Implementations . . . 67
7.2.2 Completion of Support for UFL . . . 67
7.2.3 Implementation of Additional Backends . . . 68
7.2.4 Generation of GPU Kernels . . . 68
7.2.5 Automated Exploration of Optimisations . . . 68
7.2.6 Development of Interface Code . . . 68
7.2.7 Capture of Iteration in UFL . . . 68
7.3 Manifesto . . . 68

A Optimised Kernel Library 69

B UFL Codes for Advection and Diffusion **71**
B.1 Advection 71
B.2 Diffusion 71

Chapter 1

Introduction

Fluidity [Gorman *et al.*, 2008] is a general-purpose computational fluid dynamics package developed by the Applied Modelling and Computation Group in the Department of Earth Science and Engineering at Imperial College. Explorations into using NVidia *Graphics Processing Units* (GPUs) to accelerate iterative solvers in Fluidity have been performed [Markall and Kelly, 2009, Perryman and Kelly, 2008] with results showing an order of magnitude speedup on typical hardware. However, iterative solvers are a generic part of many computational science programs, and a large research effort is devoted to using GPU hardware for their acceleration [Buatois *et al.*, 2007, Bolz *et al.*, 2005, Wiggers *et al.*, 2007, Cevahir *et al.*, 2009, Wang *et al.*, 2009]. For these reasons, it is thought to be more profitable to focus our efforts on accelerating the Assembly phase, which is specific to Fluidity.

There are several issues with using GPUs to accelerate applications. Doing so requires extensive modification of existing code. This prohibits their use for accelerating many applications, as making such modifications requires a large amount of effort. Often there is a large investment in an existing codebase, and making these modifications requires that large portions of it are obsoleted. Also, although the CUDA language [NVidia, 2007] and the NVidia Tesla Architecture [Lindholm *et al.*, 2008] are presently dominant, this will change in the future. Further extensive modifications must be made to an application each time a new architecture is targeted.

1.1 Project Outline

This project is an investigation into how Fluidity may be modified to exploit current and emerging multicore architectures by providing a hardware-independent abstraction for the specification of numerical methods, in particular the finite element method [Sherwin *et al.*, 2009]. Methods described using this abstraction will be compiled by a source-to-source translator which rewrites the abstract specification as a concrete implementation for the target architecture. This separation of the specification of numerical methods and their low-level implementation provides two key advantages:

- Methods specified in this language are "future-proofed": since the specification provides no description of the low-level implementation, re-targeting to future architectures only requires the development of a new code-generation backend.

- The development of numerical methods is eased: programming numerical methods is often tedious and error prone, requiring much code that is not directly related to implementing a new method, but is necessary in order to support its execution. Since a high-level specification prescribes the numerical method purely in terms of mathematical operations, the burden of writing low-level code is removed from the programmer. Mathematicians may rapidly prototype new numerical methods, and programmers are free to concentrate on other aspects of the software.

In this project we have focused on the NVidia Tesla architecture [Lindholm *et al.*, 2008] only. The work completed as part of this investigation divides into two stages:

1. GPU Implementations of the assembly phase for two Fluidity test programs were produced (Chapter 4). These implementations provide proof that substantial performance improvements may be obtained using GPUs for finite element assembly (Section 4.4), and form the basis of experiments to determine how further improvements may be obtained (Section 4.4.4).

2. An implementation of a prototype compiler that generates target-specific output code from a high-level specification has been developed (Chapter 5). This prototype demonstrates the feasibility of generating code for different target architectures from a single high-level specification.

1.2 Contributions

- We survey existing approaches to the automatic generation of finite element assembly code and evaluate them (Section 3.5). We also survey and evaluate the state of the art in GPU acceleration of finite element assembly (Section 3.2).

- We describe the implementation of a library of kernels that perform common operations in finite element assembly on the NVidia Tesla architecture. We present and evaluate strategies that have been used to optimise the performance of this library (Chapter 4).

- We show how this library is used to implement the finite element assembly phase for a variety of test problems, resulting in performance improvements of almost an order of magnitude over the equivalent CPU implementation on typical hardware (Chapter 4).

- We present a prototype implementation of a compiler which compiles descriptions of a finite element assembly phase (written in the *Unified Form Language* [Alnaes and Logg, 2009b]) into CUDA code which uses the kernel library previously described. We describe how the compiler converts the abstract, declarative specification of UFL into an imperative form that is used to generate CUDA code (Chapter 5).

- We evaluate the implementation of the UFL Compiler and show that this prototype demonstrates that the construction of a fully-fledged UFL compiler that supports multiple backends and optimisations is feasible and worthwhile (Section 6.3).

- We examine the Unified Form Language and show that it provides a sound platform for further research in this area due to the level of abstraction that is provides (Section 6.4).

1.3 Presentation

The research conducted throughout this project has been presented at the 8th International Workshop on Unstructured Mesh Numerical Modelling of Coastal, Shelf and Ocean Flows, under the title *Fitting the Ocean onto a graphics card: towards running ICOM on massively parallel processors* [Markall *et al.*, 2009].

Chapter 2

Background

2.1 Introduction

In this chapter we introduce the background material and concepts upon which our work is built. We begin by introducing the NVidia Tesla architecture, and the CUDA programming language used to develop software for this architecture. Subsequently, we provide a brief description of the finite element method, and define the Galerkin Projection. We go on to introduce Fluidity, and give an overview of the test problems, which we focus on throughout the remainder of the report. Finally, we discuss the Unified Form Language.

2.2 The NVidia Tesla GPU Architecture and CUDA Programming Language

The NVidia Tesla Architecture is a highly-parallel architecture made up of many minimally complex processing elements which are specialised to perform arithmetic operations. In this report we consider the most recent architecture, the GT200 architecture. Figure 2.1 shows an overview of its design.

The GT200 has 10 *Texture/Processor Clusters* (TPCs), which each consist of three *Streaming Multiprocessors* (SMs). Each SM contains various components. The operation of each component in an SM is as follows:

Compute Work Distribution. When a kernel is launched, this unit schedules individual units of work (blocks) onto each SM.

Streaming Processor. Each SP is a pipelined processor which executes instructions on scalar operands. Each SM has eight SPs that can perform integer or single-precision floating point arithmetic, and one SP which can perform computations on double-precision floating-point operands. The streaming processors also share a register file, consisting of 16384 registers, supporting the execution of a large number of threads on each SM.

Multithreaded Issue. The MT Issue unit issues instructions to each of the SPs. Since there is only one MT Issue unit, each SP in an SM must execute the same instruction concurrently.

Special Function Units. These units are specialised processors which perform mathematical operations such as sine and cosine. We do not make use of the SFUs in this work, and do not consider them further.

Caches. Each SM has an instruction cache, which is similar to the instruction cache of a typical processor. The constant cache stores data that does not change throughout the execution of a kernel.

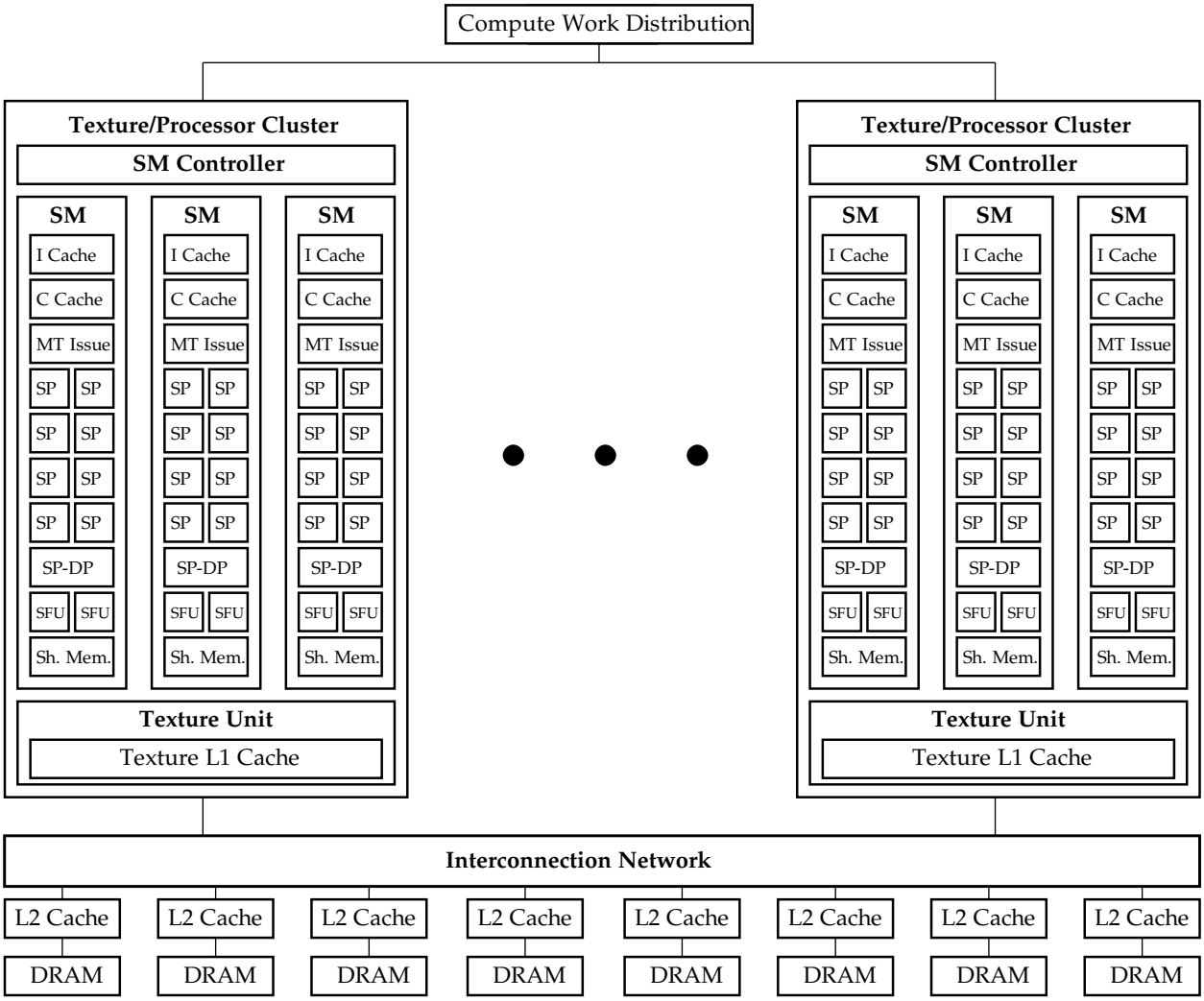


Figure 2.1: NVidia GTX280 Architecture, as in [Lindholm *et al.*, 2008]. Eight of the identical Texture/Processor clusters are omitted. SM: Streaming Multiprocessor. SP: Streaming Processor. SP-DP: Streaming Processor (Double Precision). I/C Cache: Instruction and Constant Caches. MT Issue: Multithreaded Issue.

Shared Memory. The Shared Memory is a software-controlled cache. It may be used by a programmer to store temporary data that is frequently used throughout a computation, in order to increase performance by reducing accesses to the global (main) memory.

Texture Unit. The Texture unit contains a read-only cache. Areas of memory which are bound (by the programmer) to Texture Memory are automatically serviced by this cache.

Interconnection Network. The GT200 architecture has eight banks of memory, necessitating an interconnection network to allow each SM to access data in any of the banks.

2.2.1 The Parallel Programming Model

To make use of a GPU, the programmer writes small programs called *kernels* which execute on the GPU hardware. These kernels are called by programs running on the CPU in the host machine. The program running on the host is also responsible for transferring data to and from the memory of the GPU.

The kernels are programmed using a *Single-Instruction, Multiple-Thread* (SIMT) programming model. This model allows the programmer to divide work (usually data-parallel operations) between a large number of threads. Threads are grouped at several granularities:

Warps. A warp is a group of 32 threads that are all within the same block. Threads within a warp all share the same program counter, and as a result must all execute the same instruction concurrently. In the case where threads within a warp diverge in their execution path (for example, when encountering an `if/then/else` construct), the execution of each branch is serialised. It is important to minimise divergence within a warp to obtain maximum performance.

Blocks. The next level of granularity is the thread block. Each block is mapped onto a particular SM, and has an affinity to that SM for the lifetime of the kernel. Every thread within a block has its own block-unique identifier (the Thread ID), which may be indexed in one, two, or three dimensions. The choice of the number of dimensions in the index is influenced by the algorithm being implemented. Consider an example of an image filter, which would be likely to be implemented using 2D indexing. This will allow straightforward calculation of the coordinates assigned to a particular thread based on its Thread ID.

The Grid. Only one grid exists, which contains all the thread blocks. Each block within the grid has a block-unique identifier, which has one or two dimensions.

A schematic representation of a 2D grid with 2D thread blocks is shown in Figure 2.2. Each thread and block has its unique identifier marked. The organisation of warps in this scenario is omitted from the diagram.

The choice of the number of threads within a block and the number of blocks within the grid are left to the programmer, subject to hardware limitations. Choosing the optimal number of threads per block is aided by use of the CUDA Occupancy Calculator [NVidia, 2009c], a tool which may be used to estimate the performance of different configurations. In order to ensure that each SM has enough work to do to maximise its utilisation, it is important to use a large number of blocks.

2.2.2 The Memory Hierarchy

The memory hierarchy in the Tesla architecture consists of several levels, which serve different purposes. This is in contrast to a more typical architecture, in which successive levels of the hierarchy have larger storage capacities, but increasing access latencies. A description of each level of the hierarchy follows:

Global Memory. This is a large area of memory accessible by any thread and the host. Its contents are stored in the DRAM behind the interconnection network. Accesses to Global Memory have a latency of several hundred cycles.

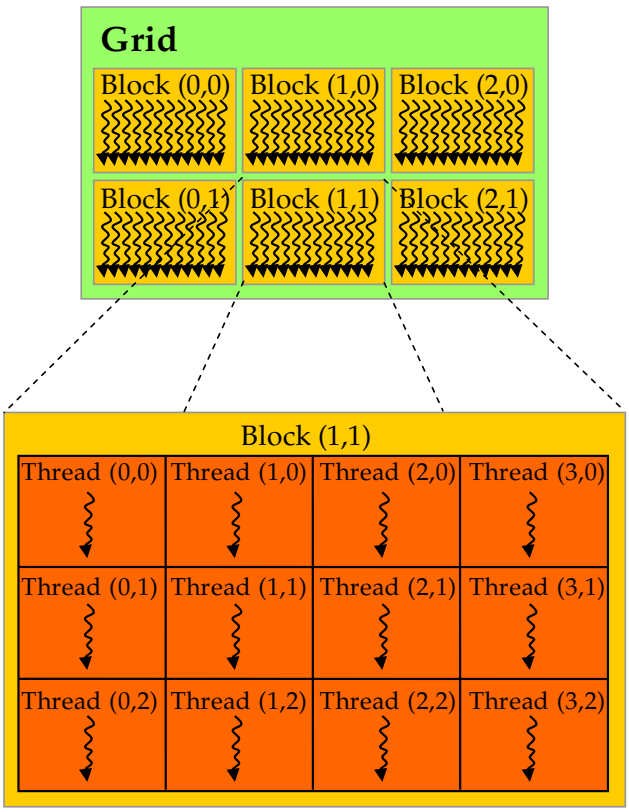


Figure 2.2: A two-dimensional grid of two-dimensional thread blocks. From [NVidia, 2007].

Registers. Local variables within kernels are stored in registers where possible. Registers may be accessed within few cycles.

Local Memory. The compiler attempts to use registers for the local variables within kernels. When a kernel has too many local variables, some of them are allocated into the DRAM. Although local memory is stored in the same hardware as the global memory, its use is transparent from a programming perspective, and only the thread owning a piece of data in Local Memory may access it. The latency of Local Memory accesses is the same as that of Global Memory accesses.

Shared Memory. The Shared Memory may be accessed more quickly than Global and Local Memories, but more slowly than registers. Threads within one block all access the same portion of Shared Memory. Pre-fetching frequently used data into shared memory may improve performance by reducing time spent waiting for accesses to Global Memory.

Texture Memory. The programmer may bind areas of data in global memory to Texture Memory. When data is accessed from these areas using a texture fetch, the texture cache is automatically used. This can increase performance for read-only data.

Constant Cache. The programmer may pre-load the constant cache with values which do not change throughout the lifetime of a kernel. This may increase performance by reducing the need to go to global memory for these constant values.

L2 Cache. The L2 cache is transparent to the programmer. Unlike a traditional L2 cache, it does not provide faster access to data when a hit occurs. Instead, the latency of the L2 cache is not significantly less than that of the DRAM, but overall performance is increased due to reduced pressure on the main memory [Volkov and Demmel, 2008].

2.2.3 Introducing CUDA

The CUDA programming language is a set of extensions to the C programming language. These extensions allow a programmer to write kernels which map on to the Tesla architecture. An API is provided that contains functions for initialising the GPU, transferring data between the host machine and the GPU, and for launching kernels.

To give a brief overview of kernels in CUDA, we consider an example. Figure 2.3 shows a naïve implementation of a kernel which computes $\mathbf{y} = \alpha\mathbf{x} + \mathbf{y}$ for a scalar α and vectors $\mathbf{x}$ and $\mathbf{y}$, written in C. The code uses a loop which iterates over each element in the vectors.

```
void daxpy(double a, double *x, double *y, int n) {
  for(int i=0; i<n; i++)
    y[i] = y[i] + a*x[i];
}
```

Figure 2.3: DAXPY Kernel in C.

To convert this function to a CUDA kernel, the loop is partitioned so that each thread takes on a portion of the work. Since each thread is within exactly one block, we can use the thread and block indices to calculate a unique portion of the vector for the thread to work on, using the `THREAD_ID` macro . This unique index is then used to initialise the induction variable for the loop. Instead of each thread incrementing its induction variable by 1, it is increased by the total number of threads. Figure 2.4 shows the CUDA implementation of a DAXPY kernel. This implementation assumes that the threads and blocks have one-dimensional Thread IDs and Block IDs respectively.

```
#define THREAD_ID (threadIdx.x+blockIdx.x*blockDim.x)
#define THREAD_COUNT (blockDim.x*gridDim.x)

__global__ void daxpy(double a, double *x, double *y, int n) {
  for(int i=THREAD_ID; i<n; i+=THREAD_COUNT)
    y[i] = y[i] + a*x[i];
}
```

Figure 2.4: DAXPY Kernel in CUDA.

It is clear from this example that getting started with CUDA is a straightforward task for a programmer already familiar with C. There are a minimal number of extra keywords in the language that allow the use of shared memory, synchronisation between threads within a block, data transfer etc., which we omit from this discussion; in-depth documentation of the API is provided in [NVidia, 2009a].

2.2.4 Remarks

Although it is very easy to start using CUDA, optimising the performance of CUDA kernels is non-trivial. Often the optimal configuration of thread and block size, and the level of the memory hierarchy to use for items of data are not obvious. To optimise an implementation, a programmer must experiment with various different choices. Debugging kernels is often tricky, with limited tools available for this purpose at present. These factors all contribute to making the optimisation process difficult and error-prone.

In addition, the programmer must write code to explicitly manage the transfer of data to and from the GPU. Making use of shared memory involves additional complexity, as data must be marshalled at the beginning and/or the end of the execution of a kernel. In general, the requirement of writing code to manage the memory hierarchy is a detriment to programmer productivity [Howes *et al.*, 2009b, Howes *et al.*, 2009a].

Since the CUDA programming language is not portable to other architectures, the work invested in optimising a CUDA program might be lost when moving to a new platform. Furthermore, future NVidia hardware may have different performance characteristics to the current generation, so optimised code may have to be re-tuned in the future.

2.2.5 Other Multicore Architectures and Languages

There are a number of other multicore architectures available or in development at present, each of which have different programming models and languages. These include the Intel Larrabee [Seiler *et al.*, 2008], the Sony/Toshiba/IBM Cell Processor [Gschwind *et al.*, 2006], and AMD's Stream architecture [Advanced Micro Devices, 2008]. It is clear that another architecture may become dominant in the future, which must be considered when deciding whether to invest in the development and optimisation of codes for the Tesla architecture.

In an attempt to standardise development for multicore architectures, The Khronos Group has developed the OpenCL Specification [Khronos Group, 2008], which describes a language that may be compiled to different multicore platforms. As with CUDA, OpenCL code must be optimised for each different target architecture.

2.3 The Finite Element Method

The *Finite Element Method* is a numerical technique for computing the solution to Partial Differential Equations. In this section, we give a brief description of the main steps in the method. A more complete introduction is given in [Sherwin *et al.*, 2009]. We begin with a general linear problem,

$$L(u) = q \tag{2.1}$$

where $L(u)$ is a linear operator, which in general consists of differential operators ($\frac{\partial}{\partial X}$, $\frac{\partial^2}{\partial X^2}$, etc.). The function $L(u)$ is referred to as the *trial function*. In the exact solution to this equation, the left-hand side is exactly equal to the right-hand side. When using a numerical method, an approximate solution is usually computed, rather than the exact solution. In this approximate solution, the LHS and RHS may not be exactly equal[1]. Denoting the computed solution u^δ, we have

$$R(u^\delta) = L(u^\delta) - q \tag{2.2}$$

where $R(u^\delta)$ is the *residual*, or error in the solution. When the computed solution is exact, $u^\delta = u$, and $R(u^\delta) = 0$. However, when a numerical approximation is produced, the form of $R(u^\delta)$ is unknown, so this term must be eliminated. To do this, we multiply the equation by a *test function*, v, and integrate over the domain Ω:

$$\int_\Omega vR(u^\delta)\mathrm{d}X = \int_\Omega vL(u^\delta)\mathrm{d}X - \int_\Omega vq\mathrm{d}X. \tag{2.3}$$

Finally we eliminate the integral of the residual by setting it equal to 0, and are left with

$$\int_\Omega vL(u^\delta)\mathrm{d}X = \int_\Omega vq\mathrm{d}X. \tag{2.4}$$

We now have the so-called *weak form* of the linear differential equation in which the definition of equality has been weakened: to satisfy this equation, instead of requiring $L(u^\delta) = q$, we only require that both sides are equal after multiplication with an arbitrary function and integration over the domain of the problem.

[1] in almost all non-trivial cases, they are not exactly equal.

2.3.1 Discretising the Domain

Although we have now relaxed the constraints for a candidate solution, finding an analytical solution to Equation 2.4 will be challenging or impossible. In most cases, we must content ourselves with an approximate numerical solution, evaluated at a finite number of points. However, the form of Equation 2.4 defines the solution to exist within a continuous functional space. Therefore, we need to find a way to convert from this infinite-dimensional, continuous space, to a finite-dimensional discrete space.

In order to achieve this, we represent the solution $u^\delta = \sum_{i=0}^{N-1} \hat{u}\Phi_i$ where N is the number of points in the discrete space, and each Φ_i is a basis function of the *trial space*. This gives us a way of approximating the infinite-dimensional trial function as a point in a finite-dimensional space. In Galerkin's method, which we consider in this project, we choose $v = \sum_{j=0}^{N-1} \hat{v}\Phi_j$. This choice makes the basis functions of the test function and the trial function the same, and the *test space* is the same space as the *trial space*. The basis functions Φ_i are defined such that

$$\Phi_i(x_i) = 1, \text{ and } \forall k : i \neq k, \Phi_i(x_k) = 0 \tag{2.5}$$

where x_i is the i-th node in a grid of discrete points in the domain. For example, in a 1D domain we might define the basis functions such that they are piecewise linear as follows:

$$\Phi_i = \begin{cases} \frac{x - x_{k-1}}{x_k - x_{k-1}} & \text{if } x \in [x_{k-1}, x_k] \\ \frac{x_{k+1} - x}{x_{k+1} - x_k} & \text{if } x \in [x_k, x_{k+1}] \\ 0 & \text{otherwise.} \end{cases} \tag{2.6}$$

We may alternatively define basis functions of higher order, provided that they are piecewise continuous and satisfy the condition given in Equation 2.5. Figure 2.5 shows that in a four-node 1D domain, there are four basis functions, as defined in Equation 2.6.

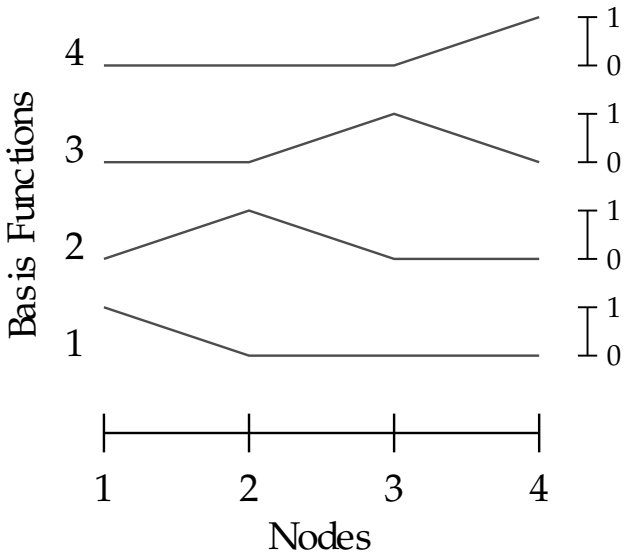


Figure 2.5: Piecewise continuous linear basis functions of order 1 over a four-node, three element one dimensional domain Ω.

2.3.2 Assembly and Solution

Having discretised the computational domain, we may now go about finding the solution in this discrete domain. This amounts to computing the projection of the solution in the test space onto the trial space. The left-hand side of Equation 2.4 defines an inner product between the test function v and the trial function $L(u)$, which may be considered as a projection of v into $L(u)$ (and

vice-versa). This projection is known as the *Galerkin Projection*. In practice, to evaluate this projection, we assemble a system of linear equations

$$A\mathbf{x} = \mathbf{b} \tag{2.7}$$

$\mathbf{b}$ is a known vector of coefficients of the basis functions of the solution in the test space. The known matrix A defines the projection from the test space to the trial space. Finally, the unknown vector $\mathbf{x}$ represents the solution in the trial space.

There is a correspondence between the right-hand side of Equation 2.4 and the vector $\mathbf{b}$. Since the function q is known, we may directly evaluate it at any point. In order to discretise the function, we treat it in the same way as the test and trial function spaces, and assume that its numerical approximation is of the form $q^\delta = \sum_{i=0}^{N-1} \hat{q}_i \Phi_i$. In the case where q is constant or linear, it is exactly represented by q^δ; otherwise, the function q^δ needs to be approximated such that $q^\delta = q$ at each mesh point. As a result, the value of q^δ is known at each node, and we can use this information to construct the right-hand side vector $\mathbf{b}$.

To assemble each row A_i of the matrix A, we integrate the left-hand side over each element adjacent to the node i. This provides a row of A coupling adjacent nodes. In practice this is achieved by integrating the left-hand side over each element, which produces an $n \times n$ matrix, called an *element-local matrix* or just *local matrix* where n is the number of nodes per element. The terms in this matrix are then added into the *global matrix*, A, at positions dependent upon the node numbers of the element.

The left-hand side often consists of integrals that cannot be efficiently evaluated analytically, so a numerical scheme such as Gaussian quadrature must be used. Since these quadrature schemes evaluate a function over a predefined interval (for example, $[1, -1]$ in Gaussian quadrature), it is necessary to compute the transformation from this reference interval to the actual physical location of each element. This transformation may be used to approximate the value of the integral over the element in its physical space. For a more complete discussion of how element-local matrices are computed, see [Sherwin *et al.*, 2009].

Having assembled A and $\mathbf{b}$, the system of equations may be solved. In most systems A is very large and sparse. Since direct solution methods are often impractical for systems of this nature, iterative solvers are preferred. When A is symmetric and positive-definite, the Conjugate Gradient (CG) method [Shewchuk, 1994] is often the most efficient algorithm for finding a solution. For other systems, the Generalised Minimum Residual (GMRES) method [Barrett *et al.*, 1994] is preferred.

2.3.3 Boundary Conditions

Often boundary conditions are a necessary condition for the existence of a unique solution to a differential equation. Boundary conditions may be classified as follows:

Dirichlet BCs. Dirichlet BCs specify the exact value of the solution at a particular boundary node. A crude way to impose a Dirichlet BC at node i is after the assembly of A, to zero out all entries of the i-th row apart from the diagonal entry, which is set to 1, and to set the i-th element of b to the prescribed value. In practice this is inefficient, and many implementations avoid the assembly of entire rows where a Dirichlet BC is applied.

Neumann BCs. Neumann BCs specify the value of a derivative at a boundary node. A characteristic of the finite element method is that if no boundary condition is explicitly applied to a boundary node, a Neumann boundary condition $\frac{\partial u}{\partial X} = 0$ is implicitly applied at that node. The application of Neumann BCs usually involves surface integration over the boundary in question, resulting in a terms which are added into $\mathbf{b}$.

Robin BCs. Robin BCs are a weighted combination of Neumann and Dirichlet BCs. In practice, they are rarely used.

In order to reduce the complexity of the implementation, the application of boundary conditions has not been included in the software developed throughout this project.

2.4 Fluidity

The Fluidity [Gorman *et al.*, 2008] code uses the finite element method to solve a wide variety of systems, including the compressible flow of Newtonian fluids in conservative and non-conservative form on unstructured, adaptive meshes. It may be used for ocean modelling, solving the incompressible non-hydrostatic Boussinesq equations, including additional terms for tidal simulation such as those representing the gravitational interaction of the Earth and Moon, and equilibrium tide and Coriolis terms. As well as supporting the modelling of a wide range of different problems, various finite element discretisations including continuous Galerkin, Petrov-Galerkin, and discontinuous Galerkin are supported, using a variety of element types. For a complete overview of these topics and a description of the code, refer to the Fluidity manual [Ham *et al.*, 2009].

A large effort has been devoted to parallelising Fluidity on distributed architectures using MPI. The code is regularly tested against experimental data and theoretical test cases [Farrell *et al.*, 2009], ensuring the correctness of the solutions it produces. It may be noted that there has a large investment of time, money and effort into the development of the Fluidity codebase.

In order to exploit multicore architectures such as the NVidia Tesla, a large portion of this investment will be obsoleted, and further effort will be required as development, testing, and tuning of CUDA code will be necessary. As stated in Section 2.2.4, the value of this additional investment might be short-lived, as exploitation of emergent multicore architectures will require further iterations of the development cycle. As a result, it is infeasible to port Fluidity to new multicore architectures using the low-level programming languages and tools that are usually provided for this purpose.

In the following two subsections, we describe the equations and their discretisations that are solved in two test programs, `test_laplacian` and `test_advection_diffusion`. We focus on these two problems throughout the rest of this project.

2.4.1 `test_laplacian`

The `test_laplacian` program is the most minimal test program in Fluidity. The purpose of the program is to solve the equation

$$\nabla^2 u = f \tag{2.8}$$

over the unit square. The Neumann boundary condition $\frac{\partial u}{\partial x} = 1$ is applied to one edge of the square. The right-hand side function defined as

$$f(x,y) = \frac{\pi}{4}\left(cos\left(\frac{\pi}{2}x\right)cos\left(\frac{\pi}{2}y\right) + \frac{\pi}{2}sin\left(\frac{\pi}{2}x\right)\right). \tag{2.9}$$

To solve this equation using the finite element method, we multiply by a test function v and integrate over the domain:

$$\int_\Omega v\nabla^2 u \mathrm{d}X = \int_\Omega vf \mathrm{d}X \tag{2.10}$$

As it is not possible to use a second derivative in a bilinear form we need to integrate the left-hand side term by parts to lower the order of the derivative. This leads to:

$$-\int_\Omega \nabla v \cdot \nabla u \mathrm{d}X + \int_{\partial\Omega} v\nabla u \cdot \mathbf{n} \mathrm{d}s = \int_\Omega vf \mathrm{d}X \tag{2.11}$$

This system may then be discretised, assembled, and solved, to find the value of u at each point in the discrete domain. In the test program, the basis functions are equivalent to those defined in Equation 2.6.

2.4.2 `test_advection_diffusion`

This test program has been written by David Ham, a member of the Applied Modelling and Computation Group, for the purpose of comparison with a CUDA implementation. The program determines the concentration of a tracer (such as saline, or dye) throughout the domain at a given time, given an initial tracer concentration, and the velocity and diffusivity throughout the domain. The following description of the method is reproduced from the documentation of this test problem [Ham, 2009b]. The system may be modelled using the advection-diffusion equation:

$$\frac{\partial T}{\partial t} + \nabla \cdot (\mathbf{u}T) = \nabla \cdot \overline{\overline{\mu}} \cdot \nabla T \tag{2.12}$$

where T is the tracer concentration, $\mathbf{u}$ is a vector representing the velocity, and $\overline{\overline{\mu}}$ is a rank-2 tensor of diffusivity. We may physically interpret the second left hand side term as one which describes the advection of the tracer, and the right hand side term as describing the diffusion, both with respect to time. Multiplying this equation by a test function q and integrating over the domain Ω with boundary $\partial\Omega$ gives:

$$\int_\Omega q \frac{\partial T}{\partial t} \mathrm{d}X + \int_\Omega q \nabla \cdot (\mathbf{u}T) \mathrm{d}X = \int_\Omega q \nabla \cdot \overline{\overline{\mu}} \cdot \nabla T \mathrm{d}X \tag{2.13}$$

Integration of the advection and diffusion terms by parts gives:

$$\int_\Omega q \frac{\partial T}{\partial t} \mathrm{d}X - \int_\Omega \nabla q \cdot \mathbf{u}T \mathrm{d}X + \int_{\partial\Omega} qT\mathbf{n} \cdot \mathbf{u} \mathrm{d}s = - \int_\Omega \nabla q \cdot \overline{\overline{\mu}} \cdot \nabla T \mathrm{d}X + \int_{\partial\Omega} q\mathbf{n} \cdot \overline{\overline{\mu}} \cdot \nabla T \mathrm{d}s \tag{2.14}$$

We fix the boundary condition to

$$\int_{\partial\Omega} qT\mathbf{n} \cdot \mathbf{u} \mathrm{d}s - \int_{\partial\Omega} q\mathbf{n} \cdot \overline{\overline{\mu}} \cdot \nabla T \mathrm{d}s = 0 \tag{2.15}$$

which implies that the domain of the simulation is perfectly insulated: no tracer will leave the domain by being advected or diffusing through the walls of the domain. A useful side-effect of this boundary condition is that it is implicitly assembled into the system of equations, simplifying the implementation. Applying this boundary condition to Equation 2.14 gives the following:

$$\int_\Omega q \frac{\partial T}{\partial t} \mathrm{d}X - \int_\Omega \nabla q \cdot \mathbf{u}T \mathrm{d}X = - \int_\Omega \nabla q \cdot \overline{\overline{\mu}} \cdot \nabla T \mathrm{d}X \tag{2.16}$$

To compute T^{n+1} (the solution at time $n+1$) given T^n (the solution at time n), we split the computation into two parts: first, we compute the solution after advection, T^a, and use this result as input to a scheme that computes the diffusion. We begin by considering the advection scheme. Although it is possible to assemble and solve the system

$$\int_\Omega q \frac{\partial T}{\partial t} \mathrm{d}X - \int_\Omega \nabla q \cdot \mathbf{u}T \mathrm{d}X = 0, \tag{2.17}$$

which models the advection, this assembly will result in a matrix that is not symmetric positive-definite. Solving this system requires the use of a method such as GMRES; however, there are no efficient CUDA implementations of this method freely available, so we are restricted to making use of the CUDA Conjugate Gradient solver [Markall and Kelly, 2009]. In order to work around this problem, we use an explicit time-stepping scheme, the Runge-Kutta method [Weisstein, 2009]. Using a fourth-order Runge-Kutta scheme to integrate T with respect to time leads to the following systems of equations:

$$\int_\Omega qT_1 \mathrm{d}X = -\Delta t \int_\Omega \nabla q \cdot \mathbf{u}^n T^n \mathrm{d}X \tag{2.18}$$

$$\int_\Omega qT_2 \mathrm{d}X = -\Delta t \int_\Omega \nabla q \cdot \mathbf{u}^{n+\frac{1}{2}} (T^n + \frac{1}{2}T_1) \mathrm{d}X \tag{2.19}$$

$$\int_\Omega qT_3 \mathrm{d}X = -\Delta t \int_\Omega \nabla q \cdot \mathbf{u}^{n+\frac{1}{2}} (T^n + \frac{1}{2}T_2) \mathrm{d}X \tag{2.20}$$

$$\int_\Omega qT_4 \mathrm{d}X = -\Delta t \int_\Omega \nabla q \cdot \mathbf{u}^{n+1} (T^n + T_3) \mathrm{d}X \tag{2.21}$$

$$\int_\Omega qT^a \mathrm{d}X = \int_\Omega qT^n \mathrm{d}X + \frac{1}{6}T_1 + \frac{1}{3}T_2 + \frac{1}{3}T_3 + \frac{1}{6}T_4 \tag{2.22}$$

where T^n is the tracer concentration at the beginning of the timestep, T^a is the tracer concentration at the end of the advection step, and $\mathbf{u}^n$, $\mathbf{u}^{n+\frac{1}{2}}$ and $\mathbf{u}^{n+1}$ are the velocity at the beginning, middle and end of the advection step. In the `test_advection_diffusion` program, the velocity at a point is constant with time. In practice, this means that $\mathbf{u}^n = \mathbf{u}^{n+\frac{1}{2}} = \mathbf{u}^{n+1}$. As in `test_laplacian`, order 1 basis functions are used for the finite element discretisation.

After the assembly and solution of these five systems of equations, we now use the solution T^a as input to the diffusion scheme. The diffusion step may be solved implicitly since the assembly of this system does result in a symmetric positive definite matrix. For this step, we assemble and solve the system:

$$\int_\Omega qT^{n+1} \mathrm{d}X = \int_\Omega qT^a \mathrm{d}X - \frac{1}{2}\left(\int_\Omega \nabla q \cdot \overline{\overline{\mu}} \cdot \nabla T^a \mathrm{d}X + \int_\Omega \nabla q \cdot \overline{\overline{\mu}} \cdot \nabla T^{n+1} \mathrm{d}X \right) \tag{2.23}$$

giving the solution T^{n+1}. Having found the solution for T^{n+1}, the entire process may be used to compute T^{n+2} and so on, until the solution at the desired timestep is reached.

2.5 The Unified Form Language

The *Unified Form Language*, or UFL [Alnaes and Logg, 2009a], originated from the research conducted as part of the FEniCS project [Dupont *et al.*, 2003, Logg, 2007]. In order to introduce the language, we must first introduce the notion of linear and bilinear forms. For example, we might introduce the notation

$$a(v,u) = \int_\Omega \nabla v \cdot \nabla u \mathrm{d}X \tag{2.24}$$

and we refer to a as a *bilinear form*, so called as it is linear in both its arguments. A linear form takes a single argument in which it is linear. As we have seen in previous sections, use of the finite element method requires the evaluation of linear and bilinear forms as in Equation 2.24, the results of which produce local matrices which are summed into the global matrix. UFL is a domain-specific language that provides the programmer with a convenient notation to express these forms, without requiring any particular details about how to implement the evaluation of these forms to be specified. The UFL compiler outputs code to evaluate these forms and perform the global assembly, freeing the programmer from the tedious and error-prone process of writing it by hand.

We will introduce the language with a small example, which specifies the assembly of the first term in Equation 2.11. The following UFL code expresses the evaluation of the form a (which we note represents the term of interest) with test and trial functions v and u as follows:

```
psi=FunctionSpace(mesh, "CG", 1)
v=TestFunction(psi)
u=TrialFunction(psi)
A=-dot(grad(v),grad(u))*dx
```

The statement `psi=FunctionSpace(mesh, "CG", 1)` specifies that `psi` is a function space defined over some mesh (how the mesh *mesh* is obtained is not discussed in this example). The parameters `"CG", 1` tell the UFL compiler that the basis functions of the functional space are piecewise continuous polynomials of order 1, as defined in Equation 2.6. `v=TestFunction(psi)` and `u=TrialFunction(psi)` specify that v is a test function and u is a trial function, which are both defined on the same mesh as `psi`. The final line specifies that the matrix A is assembled from the form in Equation 2.24.

One of the main advantages of using UFL to specify a finite element method is that a UFL specification describes the mathematical operations from a high level without describing how these operators are implemented. Since the implementation of the method is left to the UFL compiler, the user of UFL does not have to worry about the specifics of the implementation which would have to be considered when using a lower-level language such as Fortran or C++. A UFL compiler targeting a particular architecture may be tuned to generate code which is optimised for the specific performance characteristics of the that architecture.

2.5.1 UFL Compiler Optimisations

As a UFL Compiler is given a declarative specification rather than an imperative one, it is free to make choices about how the generated code should implement the specification in order to optimise performance. We shall consider an example of one of the choices the compiler may make. It is not always efficient to assemble a matrix whenever the assignment of a bilinear form is encountered. The optimal choice depends on how the resulting matrix is subsequently used. For example, consider a portion of the assembly of the diffusion scheme described in Equation 2.23:

```
M=p*q*dx
rhs=action(M+0.5*d, t)
A=M-0.5*d
```

In this example the matrix M is not used as a matrix of coefficients in a solve (though this is not evident from the above code), but is only used as an intermediary matrix for the construction of the matrix A, and the right-hand side vector. Assembling a full sparse matrix for M will be very costly in terms of memory usage, and possibly in terms of computation required to construct the matrix sparsity pattern.

Since M is not directly required, an efficient schedule for executing this code may consist of fusing the loops that assemble the right-hand side vector and the matrix. After this optimisation, the elemental submatrices that make up the matrix M may be assembled for each element in the mesh at each iteration of the loop - this elemental submatrix may then be used to compute M's action on the elemental sub-vector that contributes to the right-hand side vector, and also added into the elemental submatrix calculation for A. At the end of an iteration of the loop, the elemental submatrix of M is no longer required, and is freed. This scheme also avoids a *Sparse Matrix-Vector* (SpMV) product being computed for a very large sparse matrix, replacing it with many small, dense matrix-vector multiplications.

Whether it is more efficient to fully assemble M or to only ever assemble its elemental submatrices may not be determined from simply examining the two possibilities, but is instead dependent upon the target architecture. In this example we seek not to demonstrate which algorithm is better, but that there is an optimisation space to be explored, and further that the user of UFL need not consider this optimisation space.

2.6 Summary

We have examined the NVidia Tesla Architecture and CUDA programming language, and it has been seen that performance optimisation of CUDA code is time-consuming and error-prone. The finite element method has been discussed in brief, and we have seen that it is a very flexible method, but its implementation is complex as it consists of many steps. The Fluidity code has

been introduced, and we have described the two test problems that we focus on throughout this report. Finally, we have introduced the UFL language, which allows finite element methods to be specified at a high level, and isolates the description of a method from its low-level implementation.

Chapter 3

Related Work

3.1 Introduction

In this chapter we seek to examine and evaluate work related to that conducted as part of this project. We draw on research in the following areas:

Recent developments in the implementation of finite element assembly on GPU hardware. The application of these methods includes the simulation of hyperelastic materials, the implementation of the Discontinuous Galerkin method to solve electromagnetic scattering problems, soft tissue modelling for surgical simulation, and modelling of seismic waves in earthquakes.

Generative programming. We describe the Tensor Contraction Engine and the motivation for its creation. We examine PyCUDA, a tool that may be used to automate the exploration of the optimisation space of CUDA kernels. Finally, we describe the work on automation and code generation of finite element methods, focusing particularly on the FEniCS project.

3.2 Finite Element Assembly on GPUs

3.2.1 The Genesis of the Finite Element Method on Graphics Processors

The first implementation of the finite element method using graphics processing hardware was presented in [Rumpf and Strzodka, 2001], which implemented a nonlinear diffusion scheme. At the time at which this work was done, graphics hardware was not designed with general purpose computation in mind. In order to overcome the limitations of the hardware, mathematical operators had to be encoded as operations on textures. Due to the hardware limitations in the early graphics hardware, the implementation presented was slower than an equivalent CPU implementation executed on an SGI Onyx2 with 4 195MHz R10000 processors.

Since modern GPUs are far more powerful than the hardware used in this study, and have general purpose programming interfaces, the technical contributions of this work are largely unrelated to current techniques for the implementation of the finite element method on GPUs. We shall see in the following subsections that performance improvements may be obtained when using modern GPUs for finite element assembly.

3.2.2 Hyperelastic Material Simulation

It has recently shown that finite element assembly of the system of equations for the modelling of a hyperelastic material [Ogden, 1997] can yield speedups of up to 15 times over a CPU implementation [Filipovic *et al.*, 2009a, Filipovic *et al.*, 2009b]. In this work, an implementation of the assembly of equations modelling a St. Venant-Kirchhoff material using an NVidia 280GTX GPU is presented. The authors identify that the operations which make up the assembly process are massively parallel (since each element-local matrix may be computed in independently) but the

granularity of these operations does not have an ideal mapping on to the NVidia Tesla architecture. The operations are referred to as *medium-grained*, since they are too large to be efficiently computed by a single thread, but too small to be efficiently computed by an entire thread block.

In order to implement these medium grained operations, an implementation is described in which small numbers of threads perform an algebraic operation for an element in parallel and store the intermediate result into shared memory. The use of separate kernels for each mathematical operator allows the programmer to increase efficiency by choosing the optimum number of threads to compute a result for each element on a per-operator basis. However, the negative side-effect of this approach is that the implementation quickly becomes memory bandwidth-limited, as a large number of intermediate values need to be passed between each kernel through global memory.

A proposed solution to overcome this bottleneck involves fusing kernels and passing intermediate results in shared memory. This results in a performance gain as the pressure on the memory bandwidth is reduced. An example of this optimisation is shown in Figure 3.1. On the left, the unmodified kernels pass data through global memory. The output of kernel O_1 is stored in global memory and read by kernel O_3. To create the right-hand implementation, these two kernels are fused and the output from O_1 is stored in shared memory, which is read back by the portion of the new kernel that performs the operation O_3.

The example shows that fused kernels may have used different numbers of threads to compute the result for a single element. This can lead to inefficiency as in the fused kernel, some of the threads will be idle whilst the reduced number of threads computes the work of one of the original kernels. As a result, some kernel fusions may result in less efficient code than that which executes each operation in a separate kernel and passes intermediate results through global memory.

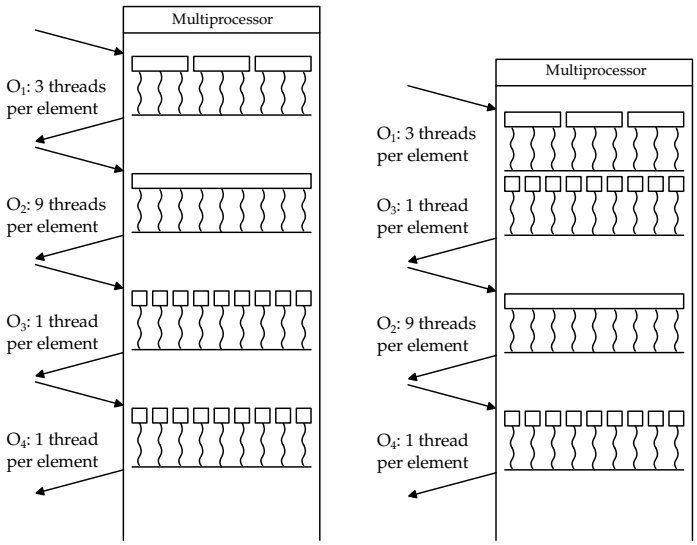


Figure 3.1: Left: Individual kernels exchange data in global memory. Right: Fused kernels exchange data through global memory. From [Filipovic *et al.*, 2009b]

When performing kernel fusions, there are also other tradeoffs that must be made to produce the most efficient implementation. It may be expected that increasing the number of operations performed by one kernel should result in an increase in speed due to the reduced memory bandwidth requirement. However, as the size of a kernel increases, its use of registers and shared memory also increases. This decreases the occupancy of the kernel, which can have a negative impact on performance. Secondly, the order in which operations are composed whilst remaining semantically equivalent has an effect on the resource usage of kernels. This design space must also be explored in the search for optimal performance. Although the authors do not discuss the automated exploration of these optimisation spaces, we observe that manual exploration is a time-consuming and error-prone process, unlikely to discover the optimum with a reasonable amount of effort.

The performance results presented by the authors show that the GPU accelerated implementation gives a speedup of 15 times over the CPU implementation when using an NVidia 280GTX GPU and and Intel Core2 Quad Q9550 with 8GB RAM. The performance improvement from fusing kernels yields a further improvement of 1.8 times over the unfused kernels. Unfortunately the authors do not report the compilers they used, or even whether computations are performed in single or double precision. Furthermore, the speedups presented are over a single core - benchmarking the CPU implementation using multiple cores would provide a more representative baseline against which the GPU implementation may be evaluated.

In conclusion, it is clear that increased performance may be obtained by using GPUs for finite element assembly, though the magnitude of this performance increase is unclear. Ideal mapping of the assembly algorithms on to the GPU hardware may be achieved by a careful choice of the number of threads per element for each operation. Optimisation of a GPU implementation will require testing the fusion of different combination of kernels, a task which ideally would be performed by an optimising compiler.

3.2.3 Discontinuous Galerkin on GPUs

Discontinuous Galerkin (DG) methods [Donea, 2003] are a class of finite element methods in which the majority of computation is applied locally to each element, and a *flux function* couples neighbouring elements. This structure of the method maps well on to a GPU architecture, since it has a high ratio of computation to data, and each element may be processed individually for most operations. The method has been implemented on the NVidia Tesla architecture and benchmarked against a CPU implementation [Klöckner *et al.*, 2009].

The DG Method

We shall briefly summarise the DG method presented in the paper, which solves a linear hyperbolic system of conservation laws:

$$\frac{\partial u}{\partial t} + \nabla \cdot F(u) = 0 \tag{3.1}$$

where F is the flux function. The system is solved on a 3D mesh of tetrahedral elements by timestepping the discrete equation

$$\frac{\partial u}{\partial t} = -\sum_{\nu} D^{k,\partial\nu}[F(u^k)] + L^k[\hat{n} \cdot F - (\hat{n} \cdot F)^*]|_{A \subset \partial D_k} \tag{3.2}$$

where ν is the set of elements, u^k is the values of u on the element k, $\hat{n}$ is the outward pointing unit normal on the element face, $(\hat{n} \cdot F)^*$ is some numerical flux in the normal direction, and D_k is the domain of element k. The Differentiation and Lifting matrices, $D^{k,\partial\nu}$ and L^k are defined by

$$M^k_{ij} = \int_{D_k} \Phi_i \Phi_j \mathrm{d}X \qquad \text{(Mass)}$$

$$S^{k,\partial\nu}_{ij} = \int_{D_k} \Phi_i \partial_{x_\nu} \Phi_j \mathrm{d}X \qquad \text{(Stiffness)}$$

$$M^{k,A}_{ij} = \int_{A \subset \partial D_k} \Phi_i \Phi_j \mathrm{d}s \qquad \text{(Face Mass)}$$

$$D^{k,\partial\nu} = (M^k)^{-1} S^{k,\partial\nu} \qquad \text{(Differentiation)}$$

$$L^k = (M^k)^{-1} M^{k,A} \qquad \text{(Lifting)}$$

The DG method may be decomposed into four stages:

Flux Gather. This involves computation of the term $[\hat{n} \cdot F - (\hat{n} \cdot F)^*]|_{A \subset \partial D_k}$. This is the only operation in which the computation is not purely element-local.

Flux Lifting. This involves the computation of the element-local Lifting matrix L^k and its action on the term computed in the Flux Gather stage.

Flux Evaluation. This is the evaluation of the Flux function $F(u^k)$ on each element k.

Local Differentiation. This stage involves computation of the local differentiation matrix $D^{k,\partial v}$ and its action on the Flux function computed in the Flux Evaluation stage.

The results of the Flux Lifting and Local Differentiation stages are summed into the matrix A (as in Equation 2.7). Since the all nodes are only local to a single element, the structure of the resulting matrix is blockwise diagonal with no coupling between blocks [Donea, 2003, p.125]. This makes the matrix very easy to invert, and this process may be done on an elementwise basis.

GPU Implementation and Optimisation

The implementation of the method is mapped on to the GPU as four kernels, each corresponding to one of the stages described in the previous subsection. We briefly outline some of the design decisions and optimisations described in the publication, and avoid a restatement of the entire description of the implementation.

It is stated that using the shared memory for storing element-local matrices leads to inefficiency since it is too small to store matrices for a large number of elements. In the case of tetrahedral elements with order 1 basis functions, each local matrix is a 4×4 matrix. The storage of one matrix in single precision requires 64 bytes, so up to 256 element-local matrices may be stored on the shared memory of one SM. Assuming one thread computes the output for a single element, this allows 256 threads to occupy a single SM. This is a reasonable level of occupancy; however, for the higher-order basis functions used in this implementation, the matrices quickly become too large. For example, a tetrahedron with order-4 basis functions has 35 degrees of freedom, so the local matrix is 35×35. In this case, each local matrix element requires 4900 bytes of shared memory, permitting the storage of only 3 element-local matrices in the shared memory of one SM.

Since the nodal data only requires a small amount of data for each element, the authors decided that it may efficiently be prefetched into shared memory at the beginning of kernel execution, and copying results back to main memory after computation has been performed. As the length (in bytes) of element data for a single element is often not a factor of 16 or 32 (the size of a half-warp and warp respectively), consecutive elements are placed adjacently in memory until their combined length is close to 64 bytes, and padding is added. See Figure 3.2 for an example of this layout. This layout allows coalesced accesses to be made when prefetching the data into shared memory, whilst wasting a minimal amount of memory space and bandwidth.

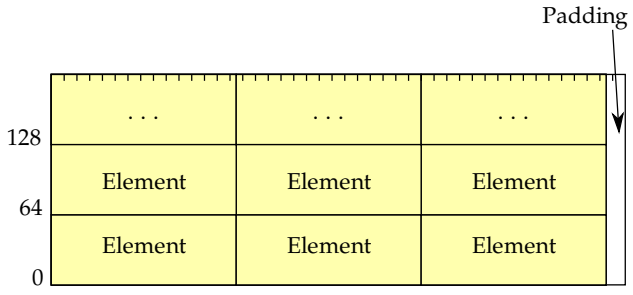


Figure 3.2: Padding of packed multiple elements to 64 bytes. From [Klöckner *et al.*, 2009].

The authors explore the idea of fusing kernels. For example, the Gather and Lifting kernels may be candidates for fusion since the output of the Gather stage is the input to the Lifting stage. However, it is concluded that this would lead to inefficient kernels, since different numbers of threads are used to compute each output value in the two kernels, leading to some threads idling for a portion of the fused kernel execution. This is in contrast to the results presented in

[Filipovic *et al.*, 2009a], where an increase in speed is reported even when kernels which use different numbers of threads per output are fused. We conclude that fusion of kernels should be explored when optimising a CUDA implementation of the finite element method, since it may lead to increased performance in some cases. However, in other cases it may be detrimental to performance.

The Flux Gather stage requires flux data to be fetched for each face of each element. Since neighbouring elements share a face, the data for each face is required to be loaded from global memory twice (see Figure 3.3). In order to reduce the memory bandwidth usage, a partitioning scheme is used where small partitions of elements are created using a greedy algorithm. In the Gather kernel, an entire partition is prefetched into shared memory before computation is performed. As a result of this optimisation, data for faces on the interior of the partition are only transferred once.

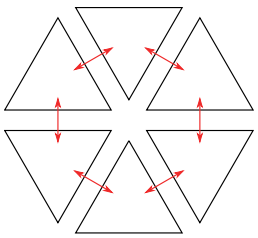


Figure 3.3: Flux between neighbouring elements.

Other optimisations that were explored include loop unrolling and constant folding, and using the texture cache to access the lifting matrix, as an alternative to using shared memory. The exploration of some of these optimisations was facilitated by using the metaprogramming system in PyCUDA (see Section 3.4).

Performance Results and Conclusions

Benchmarking of the implementation is performed using an NVidia 280GTX GPU and a single core of an Intel Core 2 Duo E8400, both operating in single precision. The test problem is an electromagnetic scattering problem, which involves finding solutions to Maxwell's Equations [Monk, 2003]. A speedup of between 24 and 57 times is shown, depending on the order of the method used. These speedups appear impressive - however, the compiler and flags used for the CPU implementation are not mentioned, and we may suspect that the CPU implementation is running sub-optimally. Additionally, benchmarking using a single core of a CPU does not provide a representative baseline to compare against, since modern CPUs have up to eight cores.

In conclusion, we have seen that the DG method is amenable to speedup using GPUs, and may form part of our further investigations into the implementation of the finite element method on GPUs. Since the optimisations that were reported to have a beneficial effect on performance differ from those described in the previous section, we conclude that optimising the implementation of finite element methods on GPUs requires a careful choice of techniques, and that there is no clear strategy for determining which optimisations may be the most worthwhile to implement. Instead, experimentation must be performed to determine which optimisations result in performance improvements for a given implementation.

3.2.4 Soft Tissue Modelling in the SOFA Framework

The *Total Lagrangian Explicit Dynamics* (TLED) method [Miller *et al.*, 2007] is a finite element method designed for modelling soft tissue for surgical simulations. The execution of the algorithm consists of two main loops. The first loop iterates over each element in the mesh to compute stress and strain forces, and the second loop iterates over nodes in the mesh, computing their new positions based on these forces. This algorithm differs from the formulation of the finite element

method described in Section 2.3, which consists of an assembly phase and a solver phase. However, the computation of the stress and strain forces is a similar process to the assembly phase. A full explanation of the TLED method is outside the scope of this report.

The first implementation of the TLED method was written in the NVidia Cg [NVidia, 2009b] language [Taylor *et al.*, 2008, Taylor *et al.*, 2007], and executed on older GPU architectures, before the availability of the Tesla architecture and CUDA. This implementation showed speedups of 16.4 times over the CPU for a model problem, and up to 14 times for a more realistic model of displacements in a model brain.

The SOFA Framework

A subsequently-developed implementation of this algorithm was produced using the *Simulation Open Framework Architecture* (SOFA) [Allard *et al.*, 2007]. SOFA is a framework designed to facilitate the development of real-time simulations, in particular medical simulations. A complete simulation is produced by composing individual behaviour models that contribute to different aspects of the simulation. For example, one may combine a collision model, a visual model and a haptic model to produce a simulation of a surgical procedure.

Each modelling area is further subdivided in a domain-specific decomposition. We focus on the behaviour model, which is used to simulate the physical motion of tissue, and has the following subdivisions: Degrees of Freedom, Mass, Force Field, and Solver. A component from each of these divisions is selected in order to construct a complete behaviour model. Implementing a new biomechanical model (such as the TLED method) requires a new Force Field module to be developed in C++. The framework provides the programmer with a relatively high-level interface to assist the development of their model, but it does not provide any abstractions from implementation in the C++ language.

Implementation of the TLED method using SOFA and CUDA is described in [Comas *et al.*, 2008]. Use of the CUDA language instead of Cg facilitated the use of shared memory and the texture cache to improve performance, though the specific details of optimisations are omitted. Pinned memory is used to increase the data transfer speed to the GPU.

Performance Results

Benchmarking the SOFA implementation using a GeForce 8800GTX GPU shows a speedup of up to 53.6 times over the CPU implementation on an Intel Core 2 Duo 2.4GHz when using single precision arithmetic. Unfortunately the compiler used for the CPU implementation is not reported, nor is the number of cores used for the simulation. We may suspect that a single core was used to execute the CPU implementation. A speedup of 37 times for a simulation of an eye in cataract surgery is reported. The accuracy of the simulation compared to the CPU is not discussed - although some discrepancy is likely due to differences in the implementation of floating point calculations, whether the solutions are qualitatively equivalent is not mentioned.

In order to demonstrate that the overhead of using the SOFA framework is very low, the TLED algorithm is implemented as a standalone program. The execution time of the standalone program is compared to the SOFA implementation for a test problem. It is shown that the use of SOFA adds an overhead of approximately 8.4%, which is considered to be an acceptable increase due to the ease of integration with other models which the SOFA framework permits.

Conclusion

We may conclude that the provision of a framework in which finite element methods may be developed and composed with other models increases the ease with which they may be implemented. However, this does not provide portability to other architectures, since the C++ code which implements the Force Field for TLED makes calls to CUDA kernels. Additionally, making changes to an algorithm implemented using C++ and CUDA will require a non-trivial effort, since the algorithm is entwined with its implementation.

3.2.5 High-Order Earthquake Modelling

[Komatitsch *et al.*, 2009] describes the implementation of a finite element earthquake model using CUDA. We do not repeat all the optimisations described, as most of them are similar to those already examined in previous sections. Rather, we shall describe an optimisation that we have not previously discussed, involving a colouring of the elements to increase the efficiency of the assembly phase.

As described in Section 2.3.2, each row of the global matrix is constructed from contributions from a single node. When this matrix is constructed by the addition of local matrices into rows and columns based on the node numbers of each element, there is potential for data races to occur. Different threads that are working on the assembly of elements that share a node will both attempt to update the same row. These concurrent updates can lead to an incorrect result being stored into the matrix. One way to overcome this issue is to use atomic operations for the addition of terms into the global matrix; however, these operations have a high cost, and may considerably slow down execution.

An alternative to using atomic operations which prevents data races is to assign a colour to each element in the mesh such that all elements of the same colour do not share any nodes. The assembly phase is then executed once for each colour, and in each of these executions, no two rows will attempt to update the same rows since none of the elements of a particular colour share any nodes. Figure 3.4 shows an example of a small mesh with an appropriate colouring for this scheme.

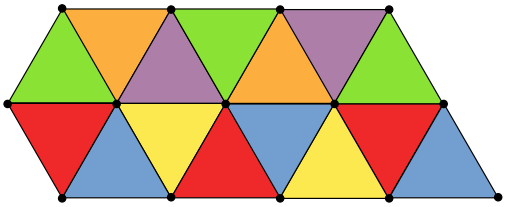


Figure 3.4: A colouring of a mesh such that no two elements of the same colour share a node.

Performance results for the implementation were gathered using an NVidia 280GTX GPU and one core of an Intel Xeon E5345. A CPU implementation is used which is compiled using the GCC compiler with the `-O3` flag. This choice of compiler and flags is justified since it is demonstrated that the binaries produced by GCC run more quickly than those produced by the Intel compilers for this problem. Speedups of between 21 and 25 are reported, for computations performed in single precision. It is shown that the results produced by the algorithm in single precision are equivalent to those produced using double precision arithmetic.

3.2.6 Finite Element in CPU/GPU Clusters

An area of research related to the implementation of finite element assembly on GPUs involves investigation into how the execution of finite element assembly may be efficiently mapped onto heterogeneous clusters composed of CPU and GPU hardware. This work is motivated by the claim that the speedups obtained using GPUs for computation in the finite element method are small enough that the contribution of the CPU to the execution time is non-trivial. Some preliminary investigations and discussion of this area are presented in [Becker *et al.*, 2009]. Since we focus on the implementation of finite element assembly on the GPU within this report rather than the efficient distribution of the workload in a cluster, we do not consider this area further.

3.3 Generating Execution Schedules for Tensor Contractions

There are often many different execution schedules for computing the result of a tensor contraction expression, which vary in the amount of memory and computation required. Consider a motivating example given in [Baumgartner *et al.*, 2002], which is the tensor expression

$$S_{abij} = \sum_{cdefkl} A_{acik} \times B_{befl} \times C_{dfjk} \times D_{cdel}. \tag{3.3}$$

The computation of this expression may be implemented naïvely using ten nested loops, requiring $4 \times N^{10}$ operations, if the range of each index is N. However, rearranging this expression gives

$$S_{abij} = \sum_{ck} \left(\sum_{df} \left(\sum_{el} B_{befl} \times D_{cdel} \right) \times C_{dfjk} \right) \times A_{acik} \tag{3.4}$$

which minimises the amount of computation required, requiring the evaluation of $6 \times N^6$ operations. However, this schedule requires a large amount of temporary storage for the results of the bracketed expressions, which may exceed the memory capacity of a target machine. Schedules which minimise the amount of computation tend to increase the storage requirements, and vice versa. Target machines differ in computational and storage capacities, and the optimal schedule depends on the availability of these resources. We see then, that the optimal expression to compute will be somewhere in between those in Equations 3.3 and 3.4.

The *Tensor Contraction Engine* (TCE) [Auer *et al.*, 2006] is a tool that may be used to search for an optimal execution schedule for the evaluation of a tensor contraction expression. Given a tensor contraction expression and the constraints of the target machine in a domain-specific language, the TCE searches for an optimal schedule and generates an implementation in Fortran.

Although the function of the TCE is not directly related to the implementation of finite element methods, we remark that it allows the user to provide an abstract specification of a mathematical operation without specifying its implementation. The TCE is free to make choices about the implementation of the operation that result in the generation of optimised code for a given target. This may be compared to the operation of a UFL compiler, which also inputs specifications of mathematical operations, and produces an implementation for a target machine as output. We conclude that a UFL compiler may also be implemented such that the execution schedule of the code it outputs is optimised for a target architecture, and that it may select optimisations that are most likely to increase performance on this architecture.

3.4 PyCUDA

PyCUDA [Klöckner, 2009] is primarily an interface to CUDA for Python. However, it may also be used for template-based metaprogramming, in order to explore some areas of the optimisation space of CUDA kernels. To make use of this facility, the programmer writes a CUDA kernel with a small amount of additional metadata. The PyCUDA system may then be used to instantiate this template kernel at runtime, generating code to explore the following optimisations:

Constant Folding. Constant folding optimisations evaluate an expression at compile time and replace any occurrences of the expression with a constant [Aho *et al.*, 2006, p.536]. This optimisation is advantageous on the Tesla architecture since it reduces register pressure. This optimisation will often be useful for replacing loop bounds in implementations of the finite element method. For example, a kernel in which threads cooperate to loop over all the elements of the mesh will have reduced register usage if the number of elements is compiled into the kernel at runtime.

Loop Unrolling. Loop unrolling reduces the number of conditional branch instructions executed as part of a loop by repeating a loop body and adjusting the bounds of the induction variable accordingly [Aho *et al.*, 2006, p.735]. Since the optimisation decreases the total number of instructions executed, a performance gain may be seen. However, this must be balanced against the increase in code size, which may put pressure on the instruction cache.

Choice of Block Size/Work Per Thread. Although the CUDA Occupancy Calculator provides some indication of grid and block dimensions that result in optimal occupancy, this is not a direct

indicator of performance. The thread block size must be chosen so as to distribute the work between threads and such that the ratio between the amount of work done by each thread and the overhead of thread creation is maximised. Since this choice is not immediately obvious, a metaprogramming system capable of generating candidate kernels with varying block and grid dimensions reduces the effort required on the part of the programmer.

In contrast to a domain-specific language, PyCUDA does not provide any abstraction from the low-level implementation details of CUDA kernels. As a result, code written using PyCUDA is limited to the NVidia Tesla architecture. Additionally, the optimisations which PyCUDA facilitates are only a small sample of many possible optimisations (see [Aho *et al.*, 2006, ch.8-12] for details of many of these). Use of a domain-specific language may provide more opportunities for optimisation, since it allows assumptions to be made that are difficult to infer from implementations in low-level languages such as CUDA.

3.5 Generative Programming/Automation of Finite Element Methods

Many tools which ease the process of programming finite element methods are available. These tools include libraries such as Deal.II [Bangerth *et al.*, 2007], Diffpack [Langtangen, 2003], and Sundance [Long, 2003], as well as domain-specific languages such as Analysa [Bagheri and Scott, 2004], FreeFEM [Hecht *et al.*, 2005] and GetDP [Dular and Geuzaine, 2005]. The level of automation of the finite element method varies between these tools. For a more complete survey of each of these packages, refer to the FEniCS lecture notes on automating the finite element method [Logg, 2007]. We have made mention of these tools to highlight the fact that there is a considerable choice in languages and libraries for implementing the finite element method. We focus on discussing The FEniCS Project [Dupont *et al.*, 2003, Logg, 2007] in more detail, for the following reasons:

- It arguably provides the user with the highest level of automation of the implementation of the finite element method compared to the other available tools.
- It provides a complete implementation of a UFL compiler.

The goal of the FEniCS project is to provide a general, efficient and simple set of tools for automating the solution of PDEs [Logg and Alnaes, 2009]. In order to demonstrate the ease with which one can write a complete implementation of a finite element method in UFL using FEniCS, Figure 3.5 shows the code required to solve Poisson's equation. The majority of this code is familiar from the description of UFL given in Section 2.5. In order to turn this into a complete implementation of a finite element solver using FEniCS Dolfin [Logg and Wells, 2009], one only needs to add extra code to define the mesh and boundary conditions, and to plot the solution.

This small piece of code contrasts heavily with the amount of code that would typically implement a solution to a similar problem: were one to implement a method to solve a similar problem without making use of a domain-specific language or library, thousands of lines of code would likely be required. However, this example is considerably more simple and compact than other implementations of the same problem that do make use of a domain-specific library. For example, the `test_laplacian` program solves a similar problem using approximately 500 lines of Fortran 90 code whilst making use of the finite element method library that is built in to Fluidity, called Femtools [Ham, 2009a].

3.5.1 Remarks

With regard to FEniCS, we highlight several points:

- Although the FEniCS tools make the development of codes that use finite element methods quite straightforward, it can be difficult to use to perform tasks that are dissimilar to those used in the finite element method.

```
from dolfin import *
mesh = UnitSquare(32, 32)
V = FunctionSpace(mesh, "CG", 1)

class DirichletBoundary(SubDomain):
    def inside(self, x, on_boundary):
        return x[0] < DOLFIN_EPS or x[0] > 1.0 - DOLFIN_EPS

# Define boundary condition
u0 = Constant(mesh, 0.0)
bc = DirichletBC(V, u0, DirichletBoundary())

# Define variational problem
v = TestFunction(V)
u = TrialFunction(V)
f = Function(V, "500.0 * exp(-(pow(x[0] - 0.5, 2) + pow(x[1] - 0.5, 2)) / 0.02)")
a = dot(grad(v), grad(u))*dx
L = v*f*dx

# Compute solution
problem = VariationalProblem(a, L, bc)
u = problem.solve()
plot(u)
```

Figure 3.5: A complete Python script to solve Poisson's Equation on a 2D domain, which defines a mesh, boundary conditions, the functional spaces, the variational problem, and plots the computed solution.

- At present, the UFL compilers included with FEniCS output code in C++ which is compiled using a standard compiler such as the GNU or Intel compilers. However, since the UFL specifications provide no implementation details, there are no barriers to the production of multiple backends for the compiler that output code for specialised target architectures, such as the Tesla architecture or the Cell processor.

- Since a complete implementation of UFL is provided as part of FEniCS, it is natural to consider whether this implementation may be integrated into Fluidity, to allow the portions of its code which implement the finite element method to be migrated from Fortran to UFL, so that the development of different UFL compiler backends eventually allows Fluidity to exploit other architectures. However, this is infeasible since the underlying data structures used in Fluidity and FEniCS have been developed separately, and as such are incompatible. Modification of the underlying data structures of either of these projects to allow their integration is a significant undertaking, as is the development of an interface layer between the two representations. These issues prohibit the effective integration of Fluidity and FEniCS.

3.6 Conclusions

We may draw the following conclusions from our literature review:

- The implementation of finite element assembly using GPUs can yield large speedups over the equivalent CPU implementations. In each of the implementations described in Section 3.2, speedups of over an order of magnitude have been obtained.

- In order to achieve maximal speedups from the use of GPUs, it is necessary to explore a variety of optimisations, including:

  - Making use of the different levels of the Tesla memory hierarchy, including shared memory and the texture cache.

  - Using different formats for packing element data together and padding the data to meet alignment requirements.
  - Mesh partitioning and colouring.
  - Fusion of kernels and array contraction.

  Since different authors report different optimisations to be beneficial to performance, and that there exist a variety of optimisations, determining the most optimal implementation of a given method is likely to involve substantial work.

- Due to the variety of finite element methods, and size of the optimisation space, the automated exploration of this space is necessary for efficient implementation of these methods on GPUs to be tractable.

- Generative programming of the finite element method based on specifications of mathematical operations that have been abstracted from implementation details permit the automated exploration of optimisations.

- In order to produce a tool to generate implementations of finite element methods from an abstract specification, preliminary work involving manual translation of a finite element method to the GPU is required. The implementations produced as part of this work may be used as a guide to the expected output from the tool, and will provide insight as to how the optimisation space should be explored.

The remainder of this report discusses the work that has been completed towards the last point. The following chapter discusses the manual translation of the two test programs, `test_laplacian` and `test_advection_diffusion`, to CUDA, whilst Chapter 5 describes the implementation of a prototype tool for generative programming of finite element methods on GPUs.

Chapter 4

Implementation of Finite Element Assembly using CUDA

4.1 Introduction

Implementations of finite element assembly routines using CUDA for the two test programs (`test_laplacian` and `test_advection_diffusion`) were produced. The motivation for producing these hand translations is as follows:

- To produce a UFL compiler that outputs CUDA code, it is necessary to have an expectation of the output it produces. Manual implementation of the test programs using CUDA provides us with reference implementations with which we may compare the generated code.
- Producing a hand-written implementation allows experimentation with different optimisations and alternate ways of structuring the code before we are committed to a firm choice about the output from the compiler. Also, we are able to use this implementation to foresee any issues with the integration of CUDA code with Fluidity.
- A pre-requisite for the development of the prototype UFL compiler is a library of optimised kernels capable of performing the operations that make up finite element assembly routines. Writing these reference implementations includes the development of such a library, which may be re-used by generated code.

The implementation of the assembly phase of `test_laplacian` was performed first, as it is the most simple test case. Later, the assembly phase for the more representative test program, `test_advection_diffusion`, was implemented. This implementation made use of most of the kernels developed for the CUDA implementation of `test_laplacian`.

The remainder of this chapter describes the implementation of the assembly routines and the optimised CUDA kernel library. Performance results are presented for the `test_advection_diffusion` implementation, which compare favourably to the CPU implementation. We do not examine the performance of `test_laplacian` in detail, as it is a very simple test case and its utilisation of the GPU is poor. This does not stand against the case for using GPUs for finite element assembly, since it is not representative of most finite element codes.

4.2 Initial Implementation of the assembly routine of `test_laplacian` using CUDA

4.2.1 The Assembly Loop in Fortran

We begin by examining the implementation of the assembly phase in its original form, which is a loop over all the elements in the mesh, making calls to the following functions:

`transform_to_physical`. This kernel implements the transformation of a basis function and its derivatives into the physical space occupied by the element, and computes the Jacobian for the transformation. Its outputs are stored in `dshape_psi`, which stores the transformed derivatives of the shape function, and `detwei`, which stores the Jacobian for the transformation multiplied by each quadrature point.

`dshape_dot_dshape`. Computes $\int_{\Omega^e} \nabla a \cdot \nabla b \mathrm{d}X$. This kernel requires `dshape_psi` and `detwei` as input. The output of this function is a local matrix.

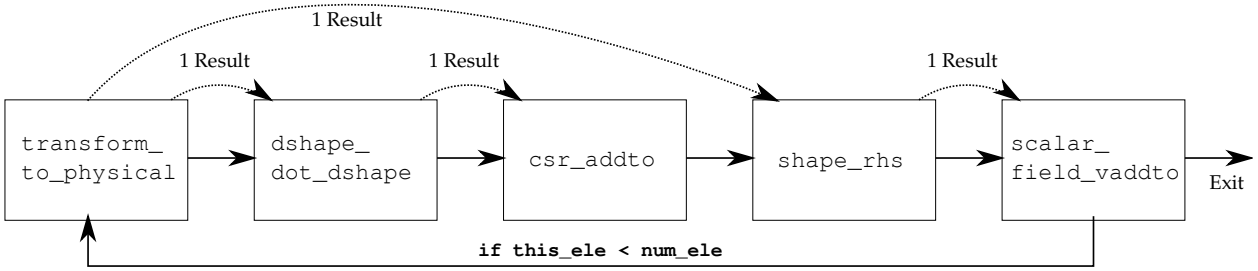
`csr_addto`. Adds a local matrix into a global matrix. In this program, the result of `dshape_dot_dshape` is added into the global matrix A.

`shape_rhs`. This computes the $\int_{\Omega^e} vf \mathrm{d}X$. This kernel requires `detwei` as input. The output of this function is an element-local vector.

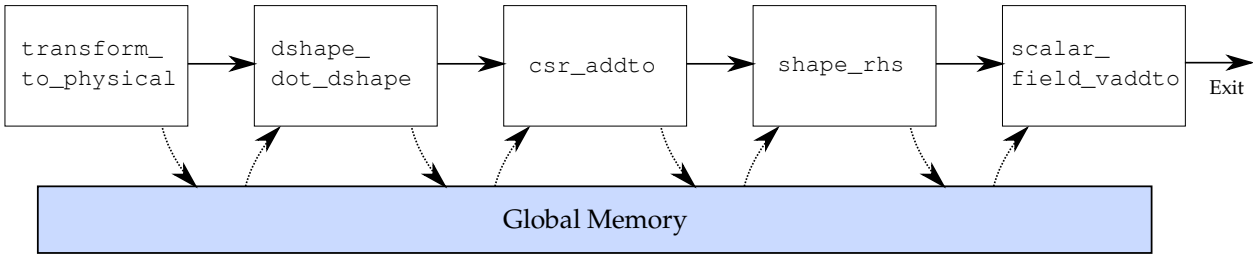
`scalar_field_vaddto`. This function adds an element-local vector into a global vector. In the test program is adds the result of `shape_rhs` into the right-hand side vector.

Each of these functions performs a suitable amount of computation to implement them as single kernels, and the resulting structure for the GPU-based assembly code partially mirrors the structure of the original Fortran source code.

The structure of the loop must be changed for the CUDA implementation. To fully utilise the GPU hardware, it is necessary to perform many tasks in parallel - a suitable level of granularity in this case is to use one thread per element, with many threads operating in parallel. Since only one kernel may execute at once on the GPU, the loop structure of the assembly phase must be flattened, and each kernel performs its computation on all elements before the next kernel may be launched. A consequence of this design is that arrays in global memory must be used to store the intermediate results for each element. Figure 4.1 shows an overview of control flow and data flow in the original and GPU implementations.



(a) Original Fortran Implementation.



(b) CUDA Implementation

Figure 4.1: Control flow (solid lines) and Data flow (dotted lines) in versions of the `test_laplacian` program.

4.2.2 Implementation of Boundary Conditions

The boundary condition affects the right-hand side vector, $\mathbf{b}$, which is the sum of two vectors:

$$\mathbf{b} = \mathbf{b}_{int} + \mathbf{b}_{bc} \tag{4.1}$$

where $\mathbf{b}_{int}$ is the contribution from nodes in the interior of the domain, and $\mathbf{b}_{bc}$ is a contribution from the boundary condition. In order to work around the need for boundary condition to be evaluated on the GPU, a scheme was implemented that uses the original Fortran code to assemble the boundary condition vector, $\mathbf{b}_{bc}$ whilst the GPU assembles the vector $\mathbf{b}_{int}$. After these computations have completed, $\mathbf{b}_{bc}$ is uploaded to the GPU, and added to $\mathbf{b}_{int}$ using a kernel that performs vector addition.

Since the work done to calculate boundary nodes grows much more slowly than the amount of work for interior nodes as the size of the mesh increases, the contribution to the total work of the boundary condition becomes negligible. Therefore, using the host to evaluate the boundary condition does not significantly affect the performance of the GPU implementation. Additionally, the boundary condition has no effect on the matrix A, and its entire assembly takes place on the GPU.

4.2.3 Translation Methodology

In order to ease the translation from Fortran to CUDA, the element assembly loop of `test_laplacian` was initially converted to equivalent C code, which is linked into the Fortran source code in its original place. This intermediate step also permitted interoperability problems between Fortran and C to be discovered and rectified without the additional complexity of working with CUDA.

Because Fortran and C are not fully interoperable [Reid, 2009], several issues were encountered during this translation phase. The following is a brief summary of the issues and their workarounds. For a more complete discussion of these issues, see [Markall, 2009].

Calling Conventions. Fortran uses the call-by-reference convention whereas C uses the call-by-value convention. C functions that are called from Fortran must be written such that all their parameters are received as pointers to values, rather than as values. Also, user-defined functions in Fortran are suffixed with an underscore by the compiler to prevent their names from clashing with any intrinsic functions. In order for the linker to correctly link the C and Fortran object files, the C function declarations must be suffixed by an underscore in the source code.

Array Indexing. Fortran arrays are stored in a column-major format, whereas C arrays are row-major. Fortran arrays also generally begin at index 1 whilst C arrays begin at index 0. In simple cases, these differences may be accounted for by subtracting one from the array index and reversing the order of indices when accessing elements of an array from a C function.

Array Descriptors. In general, Fortran allows the programmer great flexibility in handling arrays. This is achieved by using array descriptors, which are run-time data structures describing the shape of an array, its dimensions, and the range of its indices, amongst other things. Unfortunately the format of array descriptors are not part of any Fortran standard, and varies between compiler vendors. However, it was observed that when an array is passed on the stack, a pointer to the actual array data is placed in the same location that a C function expects to receive a pointer to the data. Therefore, it is possible to pass arrays from Fortran functions to C functions. In the case where a multidimensional array that varies in size is passed to a C function, additional arguments must be passed which specify the size of each dimension, and arithmetic to calculate the correct offset must be included in the C function.

Derived types and structs. Fortran derived types and C structs essentially provide the same functionality to a programmer, in that they both allow the creation of a type that aggregates storage of a number of pieces of data. Derived types and structs can be used interoperably,

provided that they do not contain multidimensional arrays, since the array descriptor for Fortran changes the alignment of all the data in the structure. In order to overcome this issue, data stored in derived types with multidimensional arrays was marshalled into a set of single-dimensional arrays before being passed to the C function.

Sub-arrays. Fortran allows a programmer to specify an operation on a subset of an array with a notation which uses the ":" symbol to denote all the elements in a particular dimension. Converting Fortran functions that use this notation to C functions requires complicated loop nests to ensure that the correct items of data are operated on, which are often non-contiguous.

In order to make the subsequent conversion from C to CUDA straightforward, each of the C kernels were written such that they consist of an outer loop which iterates over all of the elements in the mesh. This eases the translation to CUDA as this outer loop may be partitioned so that each thread performs one iteration in parallel with all the other threads. For example, the C implementation of the `csr_addto` kernel is shown in Figure 4.2

```
void csr_addto(...)
{
  for(int i=0; i<num_ele; i++) {
    for(int x=0; x<3; x++) {
      for(int y=0; y<3; y++) {
        int mpos = pos(glob_mat_findrm, glob_mat_colm,
                       node_nums[i*3+x], node_nums[i*3+y]);
        glob_mat_val[mpos]=glob_mat_val[mpos]+loc_mat[i*9+y*3+x];
      }
    }
  }
}
```

Figure 4.2: The `csr_addto` kernel implemented in C.

The parameters have been omitted for brevity. The kernel adds a 3×3 local matrix into the global matrix by looping over each element of the local matrix and adding its value into the correct location of the global matrix depending upon the node numbers of the current element. Since the global matrix is represented using the Compressed Sparse Row (CSR) [Silva, 2005] format, the function `pos` performs a bisection search to calculate the correct position in the array of values to add to. Compare this with the equivalent CUDA kernel in Figure 4.3.

```
__global__ void csr_addto(...)
{
  for(int i=THREAD_ID; i<num_ele; i+=THREAD_COUNT) {
    for(int x=0; x<3; x++) {
      for(int y=0; y<3; y++) {
        int mpos = pos(node_nums[EleIdx(i,x,n)], node_nums[EleIdx(i,y,n)]);
        atomicDoubleAdd(&global_matrix_val[mpos],local_matrix[Idx(x,y,i,n)]);
      }
    }
  }
}
```

Figure 4.3: The `csr_addto` kernel implemented in C.

In the CUDA implementation, `THREAD_ID` and `THREAD_COUNT` are macros as defined in Figure 2.4, and `Idx` and `EleIdx` are macros to calculate array indices. The atomic addition is required

since many threads are executing in parallel, more than one thread may attempt to add to the same location at once. We can clearly see the similarity between these two kernels, which demonstrates that structuring the C code such that the outer loop may easily be parallelised eases the translation to CUDA.

4.2.4 Integration with a GPU Conjugate Gradient Solver

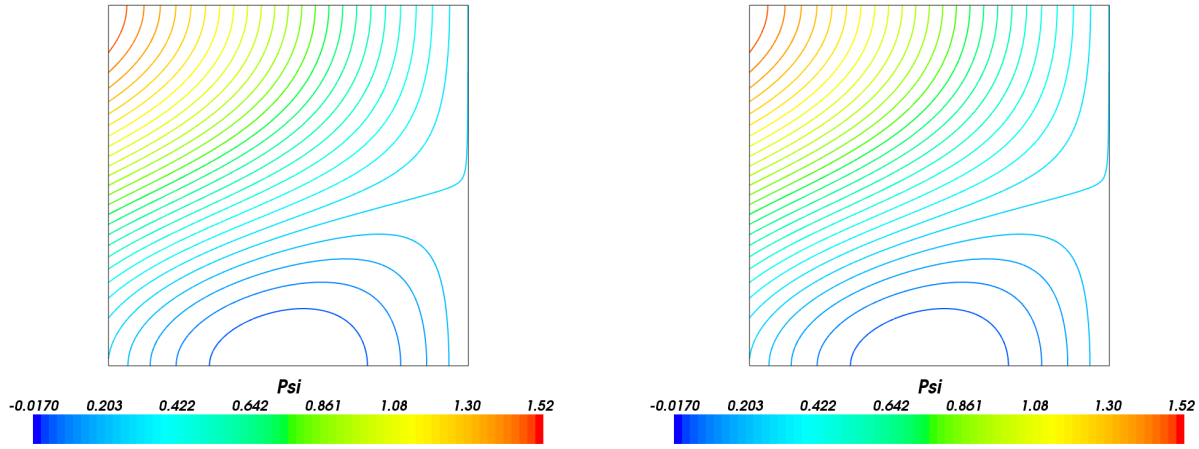
In order to obtain maximum benefit from using the GPU to perform computations, it is necessary to ensure that it is used for as many parts of the computations as possible. With a view to achieving this goal, a CUDA implementation of a *Conjugate Gradient* (CG) solver [Markall and Kelly, 2009] was used to replace the standard PETSc [Balay *et al.*, 2008] solver which is normally used in Fluidity programs.

The original Fortran source contained a call to the function `petsc_solve` which invokes the CG solver in PETSc. This call was replaced with the function `gpucg_solve`, that calls the GPU CG solver, specifying the matrix A (which is already in the global memory of the GPU) as the matrix to use for the solve. This function also takes a Fortran array, where the solution is to be stored. When the solver has completed, the solution is copied back into this array. Subsequently, control flow is returned to the original Fortran source, which outputs the solution to disk.

4.2.5 Testing

Testing the correctness of the CUDA assembly routines was performed by execution of the original and modified `test_laplacian` programs for a variety of mesh sizes, and comparing their output. `test_laplacian` provides a function for computing the total error in the solution, measured against an analytical solution. Comparison of the total error in each solution showed that the versions produce solutions with error terms equal to 7 significant figures. Since many threads are working in parallel on assembling the matrix, the order in which values are added to the global matrix will vary on each run of the assembly phase. As a result, the cumulative error in the solutions varies for each run of the CUDA implementation for digits beyond the 7th significant figure. This level of discrepancy is expected due to the non-commutativity of floating point operations, and is not indicative of an error in the program.

Figure 4.4 shows an example of the output produced by the Fortran and CUDA versions of the code. Note that the solutions appear identical.



(a) Original CPU Implementation.

(b) GPU Implementation.

Figure 4.4: Solutions computed using the CPU and GPU implementations of the assembly phase in `test_laplacian`.

4.2.6 Initial Performance Results

Once a naïve translation of the Assembly phase in `test_laplacian` had been produced, it was benchmarked against the CPU implementation to examine its performance. The execution time of the assembly phase for the GPU and CUDA implementations for varying mesh sizes is shown in Figure 4.5. We can see that the CUDA implementation requires approximately 33% less execution time than the original implementation.

Making use of a GPU is often expected to result in much greater levels of performance. In order to increase the performance of a CUDA implementation, we must attempt to discover which portions of code execute slowly, and the reasons why they do so. Use of the CUDA profiler showed that approximately 50% of the total time taken for the element assembly loop involved transferring data from the host to the GPU. It was thought that one opportunity for performance optimisation involved reducing this overhead.

One way of doing this is to try to overlap data transfer with computation. This may be achieved by converting the structure of the assembly routine in `test_laplacian` to perform operations in the following sequence:

- Initially copy enough data to the GPU to begin executing the `transform_to_physical` kernel.
- Whilst the `transform_to_physical` kernel is executing, copy data required by subsequent kernels.
- Execute the kernels `dshape_dot_dshape`, `csr_addto`, `shape_rhs` and `scalar_field_vaddto`.

The expected result of making this change is to decrease the time between beginning to copy data, and the GPU beginning execution of this data. It was hoped that copying data required by `transform_to_physical` would take only a small proportion of the total time taken to copy data, and therefore result in a substantial increase in performance. The performance of the CUDA implementation of `test_laplacian` before and after making this modification is also shown in Figure 4.4 - it is clear that this attempt at optimisation does not reduce the execution time substantially, and even increases it in some cases. Furthermore, even if this optimisation had substantially reduced the execution time in `test_laplacian`, it will not improve the overall performance of more representative problems in which assembly is performed multiple times on the same data.

4.2.7 Optimising Kernels

In order to increase the performance of the CUDA implementation, it was also necessary to examine the performance of individual kernels in order to make optimisations. The CUDA Profiler was used to examine the performance of each kernel, and the CUDA Occupancy calculator was used to estimate the effects of optimisations that reduce register usage. In general, kernels that use fewer registers allow greater SM occupancy, which allows more threads to execute in parallel, and thus increases performance. Specific optimisations which were explored include:

- Reduction in the storage of redundant data. Throughout the testing and debugging process, it was noted that several data structures store repeated data, as a consequence of the domain of the problem and the elements occupying linear space. Kernels which made use of this repeated data were modified to use only a single copy of the repeated data items. As well as reducing register usage by removing the need for an index variable, reducing the amount of data used reduces pressure on memory bandwidth. In particular, the reduced variables include:
  - `X_quad_weight` stores the weights of quadrature points for numerically evaluating integrals. In `test_laplacian`, there are three quadrature points used, which are all weighted equally. The kernels that make use of this variable used an index into an array to access the value at one of three points. To optimise the use of this variable, it is stored as one scalar value per element, and the index variable used to access it is removed.

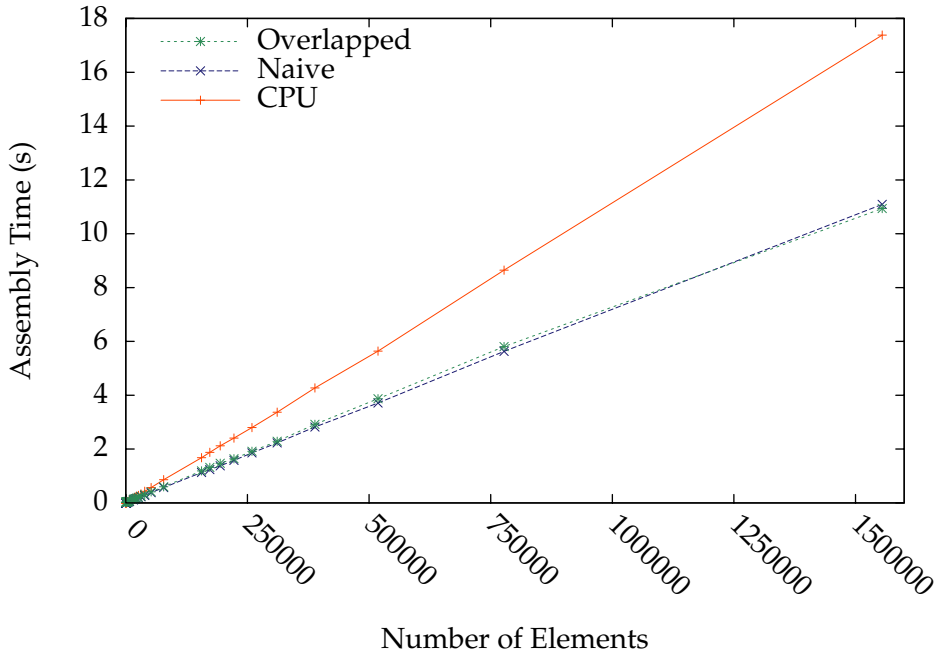


Figure 4.5: Assembly times for the CPU implementation, Naive CUDA implementation and the Overlapping CUDA implementation of `test_laplacian` for varying mesh sizes.

  - The variable `detwei` (the Jacobian multiplied by quadrature weights at each point) is also derived from `X_quad_weight` and also was stored as an array of three values per element. Its representation was also reduced to one scalar value per element.
  - The derivatives of the shape functions (`dshape_psi`) are also repeated for each quadrature point. Its representation was reduced from an 18-entry vector per element to a 6-entry vector.

- The remaining outer loops in kernels were fused where possible. These loop fusions reduce the overhead of updating the induction variable and checking the stopping condition. In the `dshape_dot_dshape` kernel, a loop interchange was necessary to make the loop fusion valid.

- Using texture memory can increase performance, particularly for memory accesses where coalescence is not achieved. The sparsity structure of the matrix was bound to texture memory, since it is accessed in a near-random pattern by the `csr_addto` kernel, which performs a bisection search when it calls the function `pos` (see Figure 4.3). The original and modified search loops are shown in Figure 4.6. In the modified version, the array access to `colm` have been replaced with a call to `tex1Dfetch` accessing `tex_colm`, which is the same area of memory as `colm`, bound to texture memory.

- *Warp Serialisation* occurs when different threads within a warp take different branches of a conditional statement. This occurs whilst performing the bisection search in `csr_addto`, since each thread is independently performing a search. Since rows of the matrix are short (in practice always less than 16 elements long), an alternative strategy involves using an entire half-warp to cooperate in performing a linear search of a row to find the correct element. However, this strategy was found to be inefficient. This is partially because rows are often very short, with only 6-7 elements. Furthermore, once the correct element in the matrix has been located, 15 threads of the half-warp are idle whilst the thread that found the correct element performs an atomic addition.

- Kernels were originally written to be capable of performing computations on two- or three-dimensional elements with varying numbers of nodes. In order to achieve this generality, the loops in the kernels have variable upper bounds; for example, a loop over the nodes

```
while(upper_pos-lower_pos>1) {
  this_pos=(upper_pos+lower_pos)/2;
  this_j = colm[row+this_pos];

  if(this_j==j)
    return this_pos+base;
  else if(this_j>j)
    upper_j=this_j, upper_pos=this_pos;
  else if(this_j<j)
    lower_j=this_j, lower_pos=this_pos;
}
```

(a) Original Implementation.

```
while(upper_pos-lower_pos>1) {
  this_pos=(upper_pos+lower_pos)/2;
  this_j = tex1Dfetch(tex_colm,
                      row+this_pos);

  if(this_j==j)
    return this_pos+base;
  else if(this_j>j)
    upper_j=this_j, upper_pos=this_pos;
  else if(this_j<j)
    lower_j=this_j, lower_pos=this_pos;
}
```

(b) Using Texture Memory.

Figure 4.6: Versions of the pos function.

of an element requires the upper bound to be equal to the number of nodes. To reduce register usage, kernels specific to triangular elements in two dimensions were produced. This allowed the variable upper bounds to be replaced with a constant, reducing register usage. Continuing the previous example, loops that iterate over the nodes of an element have an upper bound of 3 after this optimisation has been made.

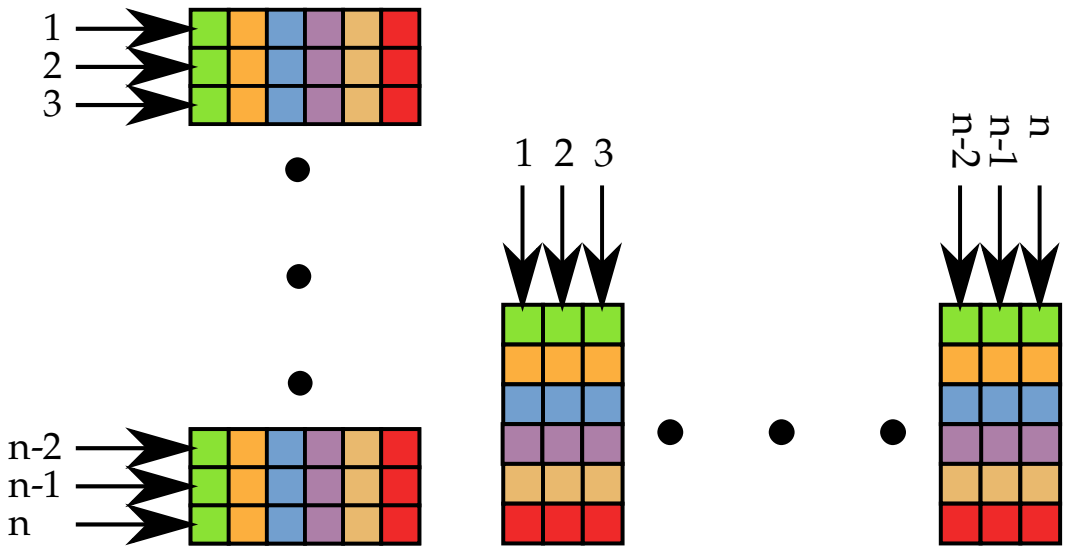
- The use of shared memory for storage of values which are re-used throughout the execution of a kernel was considered. However, in this implementation, almost all of the data that is loaded from main memory is used exactly once, prohibiting the use of shared memory to increase performance. The items of data which are reused are few enough to be able to fit into the registers, so no performance will be gained from storing them in shared memory.

4.2.8 Ensuring Coalesced Memory Accesses

To obtain high performance on the Tesla architecture, it is important to ensure that memory accesses are coalesced as much as possible. To ensure that memory accesses are coalesced, kernels must be programmed such that groups of 16 threads all access memory within a 64-byte window concurrently. Non-coalesced accesses may only utilise as little as $\frac{1}{16}$th of the available memory bandwidth.

In the naïve translation of `test_laplacian`, data which was extracted from the mesh was placed into arrays with the same ordering as in the mesh data structures. This layout prevents threads within a half-warp from performing accesses within a 64-byte window concurrently, and as a result preventing coalesced accesses from occurring. As an example of why this happens, we consider the array `X_ele`, which stores the coordinates of each node of each element. Since we are using 2D triangular elements, we have three nodes in two dimensions, which requires 6 floating-point values per element. These six values for a single element are stored adjacently in the array. When every thread within a half-warp tries to access a coordinate value for each element, addresses in memory that are not adjacent are accessed. Since the same coordinate value for each element are actually spaced six elements apart in memory, the individual threads within a warp do not access adjacent values (see Figure 4.7(a)).

In order to rectify this performance problem, we optimise this memory access pattern by transposing the data on the host before it is transferred to the GPU. The host is preferred for performing this transposition since doing so requires a scatter operation to be performed, to which the memory system of the host is far more suited. When using the resulting layout, threads which all access the same coordinate for each element now access adjacent elements in memory, resulting in coalesced reads (see Figure 4.7(b)). The transposition was applied to all of the mesh data structures in order to maximise coalescence throughout the execution of the assembly phase.



(a) Non-coalesced (b) Coalesced

Figure 4.7: Data layouts which lead to non-coalesced and coalesced accesses in kernels. Numbers indicate the Thread ID accessing each value.

4.2.9 Post-Optimisation Performance

Having made the optimisations described in the previous two sections, the performance of the CUDA implementation was again measured for varying mesh sizes. In order to determine the fraction of time taken to perform the transfer of data from the host to the GPU, the code that performs assembly on the GPU was commented out and the benchmarks run a second time. These performance results were obtained on a machine with an Intel Core 2 Duo E8400 with 2GB of RAM and an NVidia 280GTX GPU. The CPU implementation was compiled using GCC 4.4.0 with the `-O3` flag, and the CUDA implementation was compiled using NVCC 2.2. The performance results are shown in Figure 4.8.

It can be seen from these results that the copy operation now dominates the GPU version. Discounting the time taken for copying data, the GPU accelerated assembly computation is approximately 31 times faster than the CPU implementation. The time taken by the copy phase will be amortised in the `test_advection_diffusion` benchmark since it will perform many assembly steps beginning from the same initial data. As such, this large overhead is not of concern.

4.3 Implementation of the Assembly Phase of `test_advection_diffusion` Using CUDA

The main portion of the `test_advection_diffusion` program consists of a loop which computes the solution for one time step at each iteration. Given the solution at time n, the the solution at time $n+1$ is computed. The body of the loop consists of the following phases (also see Figure 4.9):

Advection Phase. Each of the systems described in Equations 2.18 to 2.22 is assembled and solved. The solution from the final solve is used as the input to the diffusion phase.

Diffusion Phase. The system described in Equation 2.23 is assembled and solved. The solution from the solve is the solution at time $n+1$, and is used as input to the Advection phase for the next iteration of the loop.

In order to make efficient use of the GPU hardware, this loop must be ported such that it runs entirely on the GPU. Transferring data back and forth between the host and GPU between each

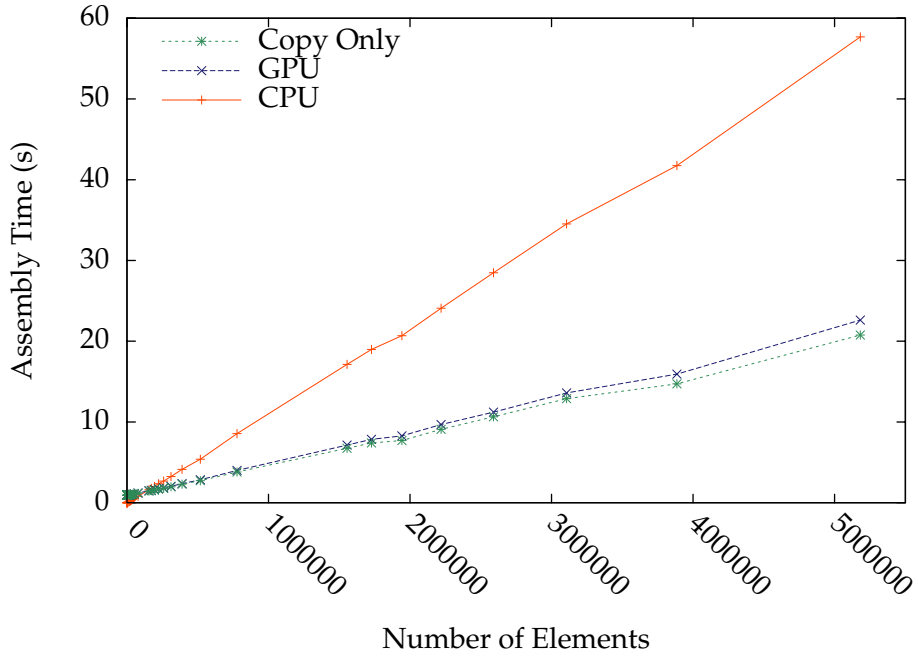


Figure 4.8: Time taken to perform assembly in `test_laplacian` with optimised coalesced kernels.

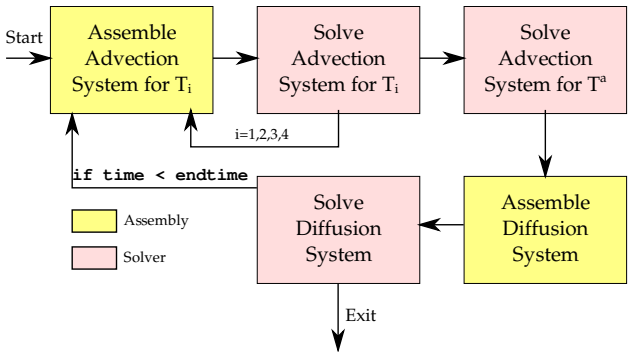


Figure 4.9: Control flow in the assembly loop of `test_advection_diffusion`.

assembly and solve is unnecessary, and will greatly slow down the execution of loop. As we have already seen, the process of copying data dominates the computation in `test_laplacian`.

The structure of the CUDA implementation is as follows. Before the loop begins execution, all the initial conditions are transferred to the GPU. The main loop begins execution, and no data is transferred back to the host during or between iterations. At the end of the computation, the solution at the current (final) timestep is transferred back to the host. Since one may be interested in the solution not just at the final timestep, but at some intermediate times, the solution at the end of specific timesteps may optionally be transferred to the host. The control flow of the CUDA implementation is shown in Figure 4.10.

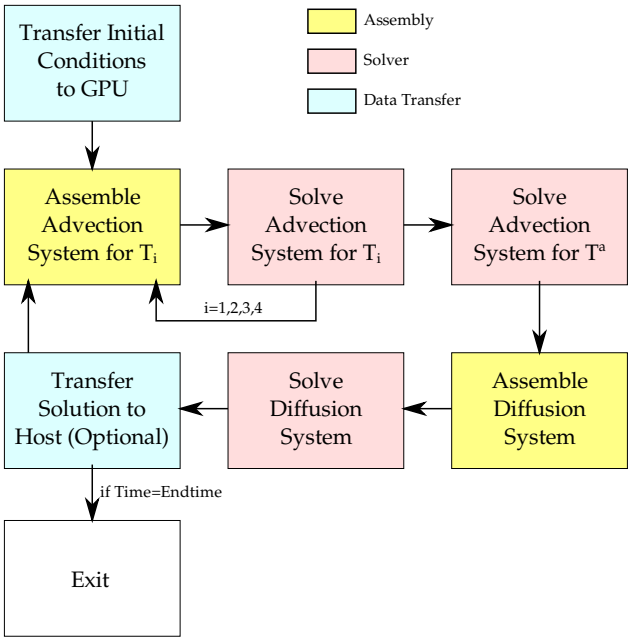


Figure 4.10: Control flow in the GPU implementation of the assembly phase in `test_advection_diffusion`.

4.3.1 Kernels Used in this Implementation

Several of the kernels that were developed whilst implementing the assembly phase in `test_laplacian` were re-used for the assembly phase in `test_advection_diffusion` . These include:

- `transform_to_physical`
- `csr_addto` - this kernel is renamed to `matrix_addto` in this implementation for consistency with other kernels performing similar operations.
- `scalar_field_vaddto` - this kernel is renamed to `vector_addto`, also for consistency with other kernels.

The implementation of several more kernels was also necessary. Required kernels which numerically evaluate integrals included:

`dshape_dot_vector_shape`. Computes $\int_{\Omega^e} \nabla v \cdot \mathbf{x} u \mathrm{d}X$ over an element e, where $\mathbf{x}$ is a vector, and v and u are test and trial functions.

`dshape_tensor_shape`. Computes $\int_{\Omega^e} \nabla v \cdot \overline{\overline{\mu}} \cdot u \mathrm{d}X$ over an element e, where $\overline{\overline{\mu}}$ is a rank-2 tensor and v and u are test and trial functions.

`shape_shape`. Computes $\int_{\Omega^e} vu \mathrm{d}X$ over an element e where v and u are test and trial functions.

Extra kernels that perform the addition of local matrices into the global matrix included:

`matrix_addto_diffusion`. This kernel sums two local matrices before adding them into the global matrix. It is required since two local matrix contributions make up the global matrix when assembling the diffusion system. The first of these is from the left-hand side term and the second is the last term in the right-hand side of Equation 2.23. This kernel is not strictly required, since the original `matrix_addto` kernel may be called once for each local matrix contribution. However, doing this is inefficient as it requires the matrix sparsity pattern to be searched twice as many times as using a single kernel.

`matrix_addto_sum_diag`. This kernel sums each row of a local matrix and adds the results into the main diagonal of the global matrix. It is required since in practice, *mass lumping* [Sherwin *et al.*, 2009, p.5-3] is used in `test_advection_diffusion`.

Other kernels that were required included:

`advection_rhs`. This kernel is used to compute the right-hand side of Equations 2.18 to 2.21. The operations involved in this computation include:

- Calculation of the divergence of a vector field, which essentially requires a matrix-matrix multiplication and a sum reduction of the result. The vector field in this case is the velocity field.
- Summation of two local matrices to produce a new local matrix.
- A matrix-vector multiplication using the new local matrix.

Since this kernel is made up of fairly generic operations, it may also have been split into kernels which perform each of these operations separately, passing these results between kernels using global memory. This may result in better performance due to decreased register usage increasing occupancy. Exploring whether doing so is beneficial is part of future work, and may be explored much more efficiently using an automated tool to explore this optimisation space.

`diffusion_rhs`. This kernel computes the right-hand side vector in the diffusion system of equations. Two steps are performed in this computation. First, two local matrices are summed. The resulting local matrix is multiplied by a local vector. As with the `advection_rhs` kernel, the most efficient implementation may consist of using two generic kernels to compute the same result. However, since this kernel is very small, it is unlikely that this will be the case.

`scatter_rhs_values`. When the data is transferred to the GPU, the host extracts data from the field containing the tracer concentration, T, for each node of each element. This data is placed into an array that stores the value at each node for each element. This storage layout results in the duplication of some data, since several elements often share a single node. This becomes an issue after the solve has taken place, since the vector of solutions contains one solution value per node. The `scatter_rhs_values` kernel performs the task of copying the solution at each node into an array that stores the solutions for each node per element. This new array is in the correct layout to be used as input to the next iteration of the loop.

It is noted that the majority of the new kernels which were required perform generic tasks in finite element assembly. Each of these kernels is likely to be re-usable for the implementation of methods to solve other equations.

4.3.2 Testing

Testing the CUDA implementation of the main loop of `test_advection_diffusion` was performed by using it to solve a model problem, and comparing its output to the original implementation solving the same problem. The model problem consists of a square domain with a vector field

specifying velocity within the domain, and a tensor field specifying the diffusivity. An initial concentration of some tracer is advected and diffused according to these fields.

In the model problem, the domain is a square with coordinates ranging from (-1.2, -1.2) to (1.2, 1.2). The velocity at a given point is is defined by the function:

$$V = \begin{cases} \begin{bmatrix} y\ cos(\frac{\pi r}{2}) \\ -x\ cos(\frac{\pi r}{2}) \end{bmatrix} & \text{if } r < 1 \\ \begin{bmatrix} 0 \\ 0 \end{bmatrix} & \text{otherwise} \end{cases} \tag{4.2}$$

where x and y are the coordinates of the point, and r the distance from the centre of the domain. This velocity field is visualised in Figure 4.12(a). Diffusivity is defined as the rank-2 unit tensor everywhere in the domain. The initial tracer concentration is specified by the function:

$$T_{initial} = \begin{cases} 1 & \text{if } r < \frac{1}{4} \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

where r is the distance from the point (-0.5,0). The solution is computed for 50 timesteps beginning from these initial conditions on an unstructured mesh with 14336 elements. The mesh is shown in Figure 4.11. Since advection and diffusion are computed separately, we may run the simulation solving for advection, diffusion or advection and diffusion. Figures 4.13 to 4.15 show the initial condition, and the expected result after 50 timesteps for each of these configurations. The initial condition appears different in each case since the scale of the legend has been matched with that at the final timestep. The errors introduced by the numerical scheme are responsible for the solution values moving outside the range [0,1].

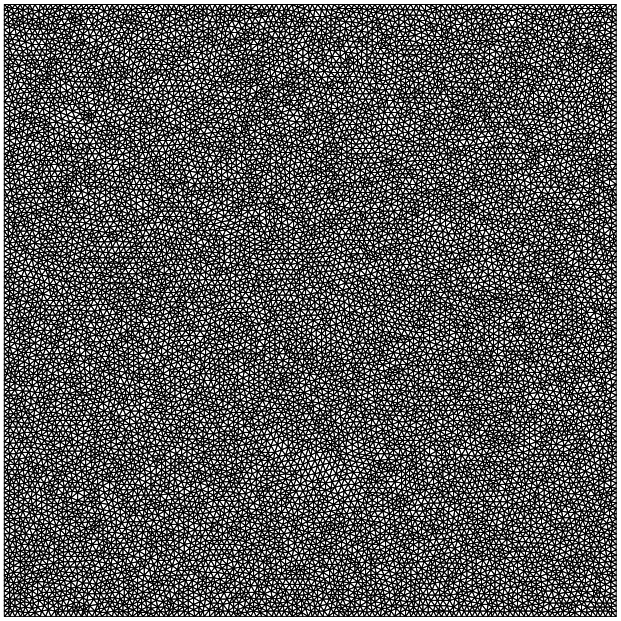
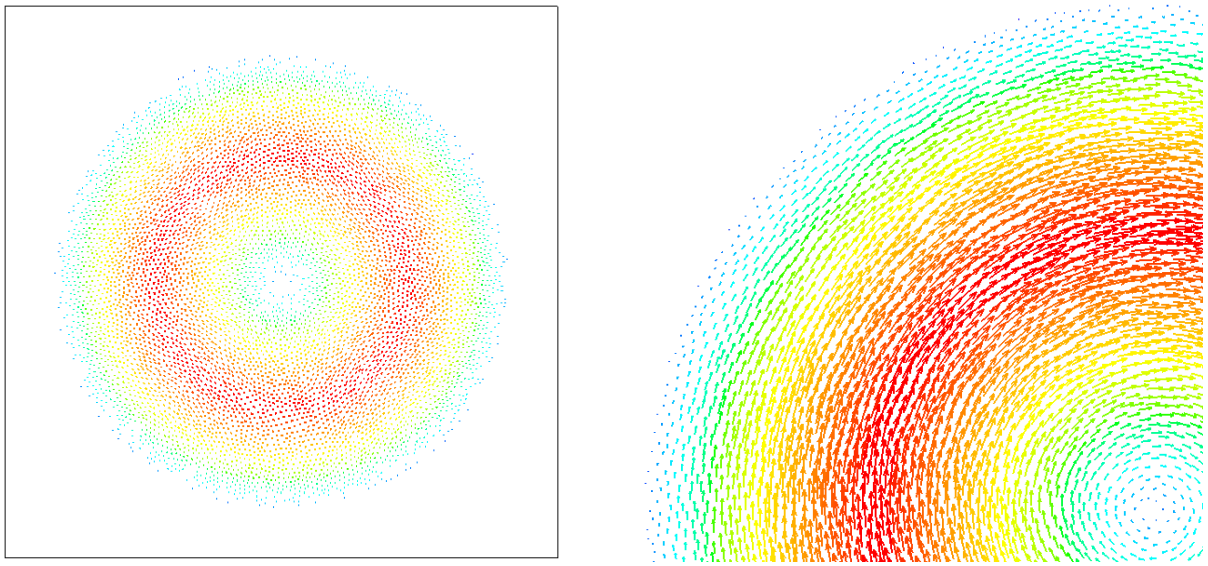


Figure 4.11: The mesh used for testing the CUDA implementation of the assembly phase in `test_advection_diffusion`.

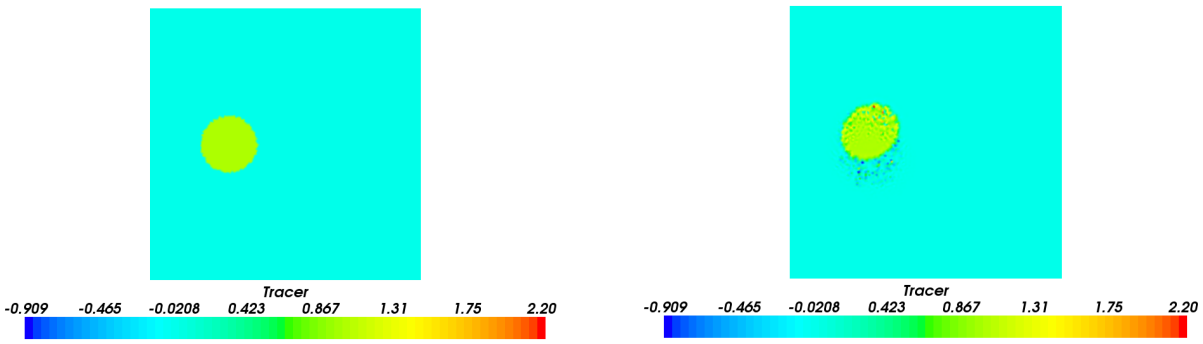
The results of testing showed the output from the CUDA implementation to agree with that from the CPU implementation to at least 6 significant figures in all cases. These small discrepancies are again expected due to the non-commutativity of floating point arithmetic. These differences are slightly larger than for `test_laplacian` since the solution at each timestep is used as the input to the next timestep, compounding the errors.



(a) Entire domain.

(b) Close-up showing direction.

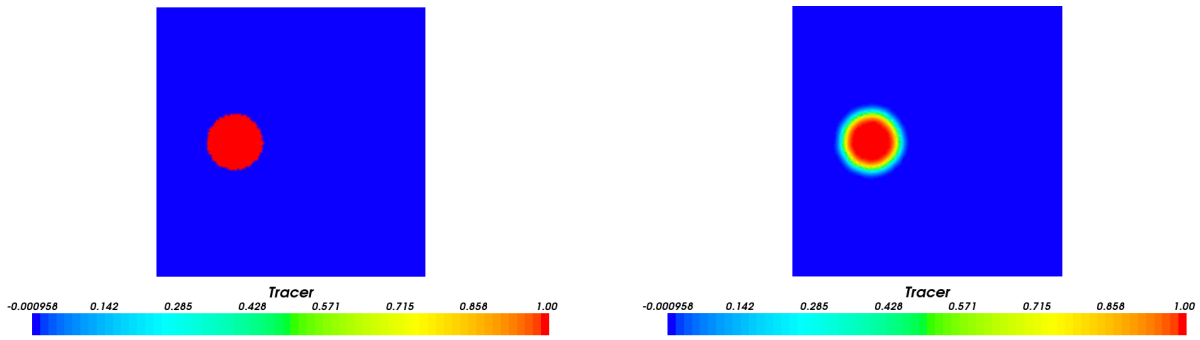
Figure 4.12: The Velocity field used in the `test_advection_diffusion` test problem.



(a) Initial Time.

(b) Final Time.

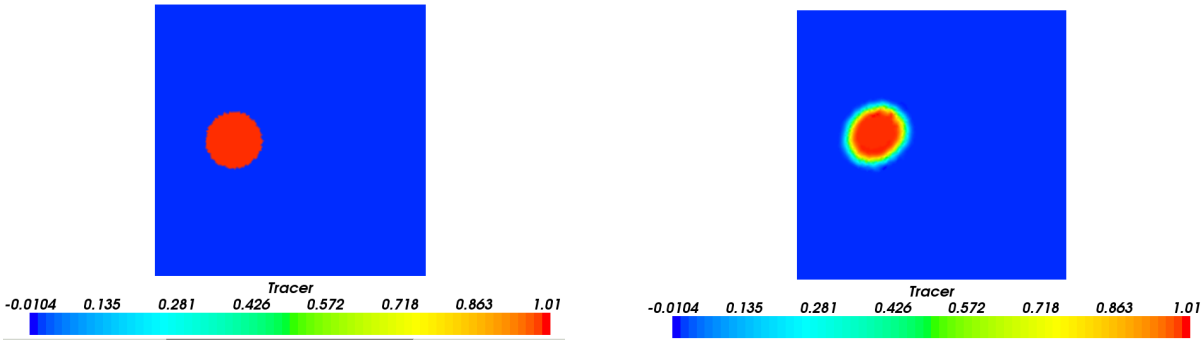
Figure 4.13: Concentration profiles of the Tracer in the advection problem.



(a) Initial Time.

(b) Final Time.

Figure 4.14: Concentration profiles of the Tracer in the diffusion problem.



(a) Initial Time.

(b) Final Time.

Figure 4.15: Concentration profiles of the Tracer in the advection-diffusion problem.

4.4 Performance Results and Analysis

Performance results were gathered using a machine with an Intel Core 2 Duo E8400 processor, 2GB of RAM and an NVidia 280GTX GPU. The Intel 10.1 C++ and Fortran Compilers were used to compile the CPU versions of the code, since they produce much more efficient code than the GNU compilers. The most recent versions of the Intel compilers, 11.1, were not used as they do not correctly compile Fluidity. The `nvcc` compiler included with CUDA 2.2 was used to compile the GPU code. Each test was run five times, and the averages of these five runs are reported. All computations were performed using double precision arithmetic, since it has been shown that the use of single precision arithmetic for the assembly phase is not sufficient [Markall and Kelly, 2009].

Versions of the code were benchmarked using a range of square meshes of increasing fineness, running the system for 200 timesteps starting with the initial tracer concentration defined in Equation 4.3. The initial time is recorded as the time before the host begins to extract the data from the mesh. The final time is recorded after the solution at the final timestep has been transferred back to the host. As a result, these timings capture all the overheads of making use of the GPU, including the extraction of data from the mesh, the transposition of data on the host, the transfer of data to the GPU, and the transfer of the solution back to the host. In order to accurately compare the performance of the computations, no intermediate solutions were transferred from the GPU to the host.

4.4.1 Performance of the Assembly Phase

In order to assess the performance of the code produced for this project, we begin by examining the performance of the assembly phase only. These times were obtained by commenting out code that calls the solver. Figure 4.16 shows the time taken for assembly of advection and diffusion, whilst Figures 4.17 and 4.18 show the time taken only to assemble the advection system and the diffusion system respectively.

The assembly of the advection system takes approximately 4.5 times longer than the diffusion system, since the explicit timestepping scheme requires the assembly of four systems, whereas the implicit diffusion scheme only requires the assembly of one system. We see that in all cases the GPU implementation is significantly faster that the CPU implementation.

4.4.2 Speedup and Throughput

We see that once the startup costs are amortised, there is a speedup of about 16 times over 1 CPU core or 8 times over 2 cores. Figure 4.19 shows the speedups of the CPU over the GPU, whilst Figure 4.20 shows the throughput of each implementation in terms of elements per second. We see that the throughput of the GPU implementation is poor in terms of throughput for the

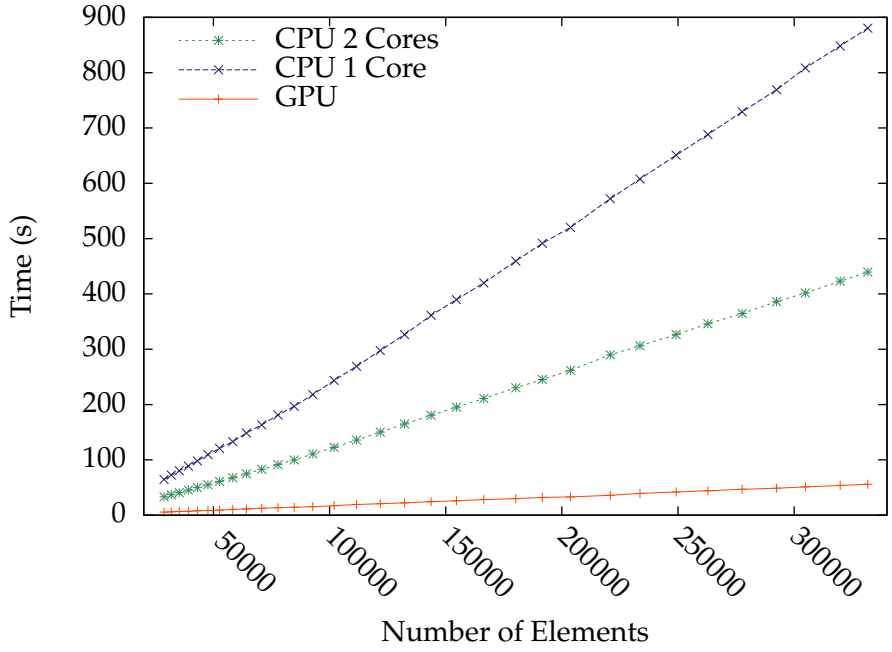


Figure 4.16: Time taken by the assembly phase in running the Advection-Diffusion system for 200 timesteps.

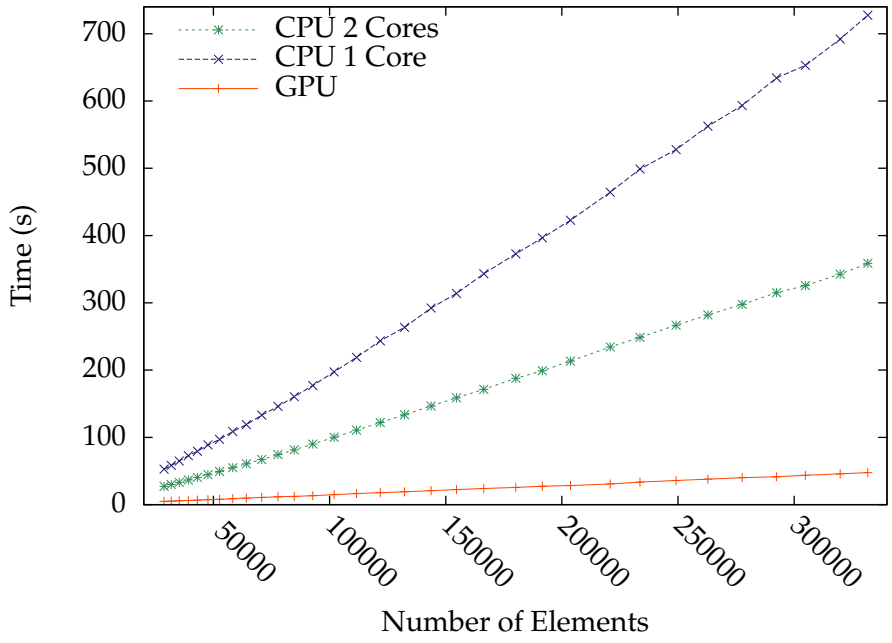


Figure 4.17: Time taken by the assembly phase in running the Advection system for 200 timesteps.

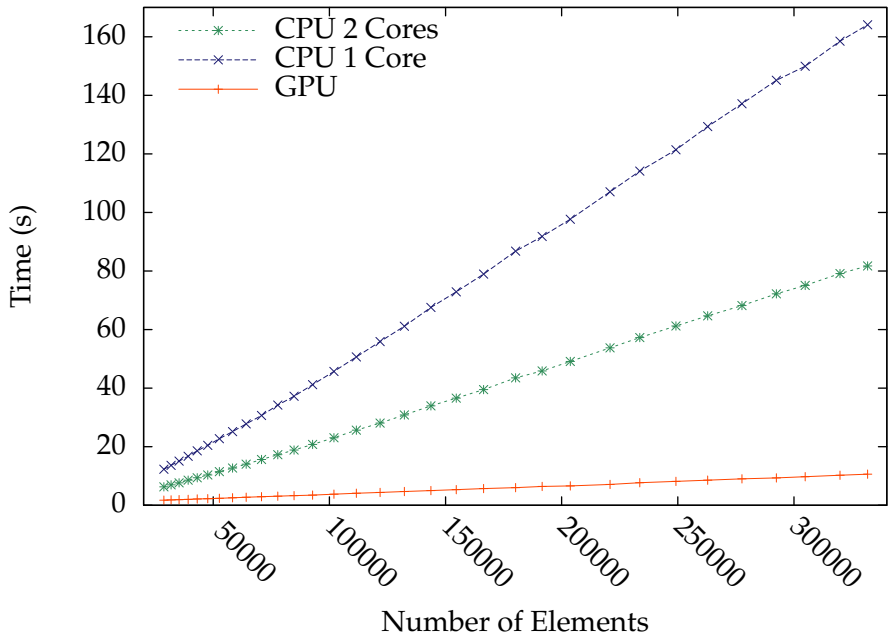


Figure 4.18: Time taken by the assembly phase in running the Diffusion system for 200 timesteps.

smaller mesh sizes. This is a consequence of the startup cost of using the GPU. This startup cost is amortised over longer runs, making the GPU favourable for larger problems.

There is a consistent speedup of 16 times over one CPU core and 8 times over two cores for the larger meshes. Although the CPU implementation was only tested on a machine with two cores, we can project that if the performance scales perfectly up to 8 cores, the GPU implementation will still be approximately twice as fast as the CPU implementation. In practice, we will not see a perfect speedup, and the return from adding more cores will diminish.

It is interesting to note that the throughput of the CPU implementations seem to decrease slightly as the problem size increases, whilst the throughput of GPU implementation increases with the problem size. We can easily attribute the behaviour of the GPU's performance to the amortisation of startup costs over long runs, including the transposition and transfer of data. We may speculate that as the problem size increases, the CPU performance may be lowered due to a reduction in cache hits.

4.4.3 Overall GPU Performance

We also examine the performance of the Assembly phase on the GPU in the context of the overall speedup obtained from using the GPU. Here, we report the total time taken to run each simulation for 200 timesteps, which includes the time taken to perform the assembly and the solution phases. Figures 4.21 to 4.23 show total simulation time for the CPU and CUDA implementations solving the advection-diffusion, advection, and diffusion systems.

Figure 4.24 shows the speedups obtained from using the CUDA implementation for the entire simulation. We see an overall speedup slightly less than that obtained for the assembly phase alone, due to the lower performance of the CG solver. There is reason to believe that the performance of the CG solver can be improved [Markall and Kelly, 2009], but this has not been the focus of this project.

Since we achieve these speedups for the assembly and solve phases, it is clearly a worthwhile task to further investigate the use of GPUs for the implementation of the finite element method. We believe that the implementation we have produced has the potential for further performance increases. In the next subsection we examine the performance of the assembly phase to determine where bottlenecks exist and suggest optimisations that may reduce their impact.

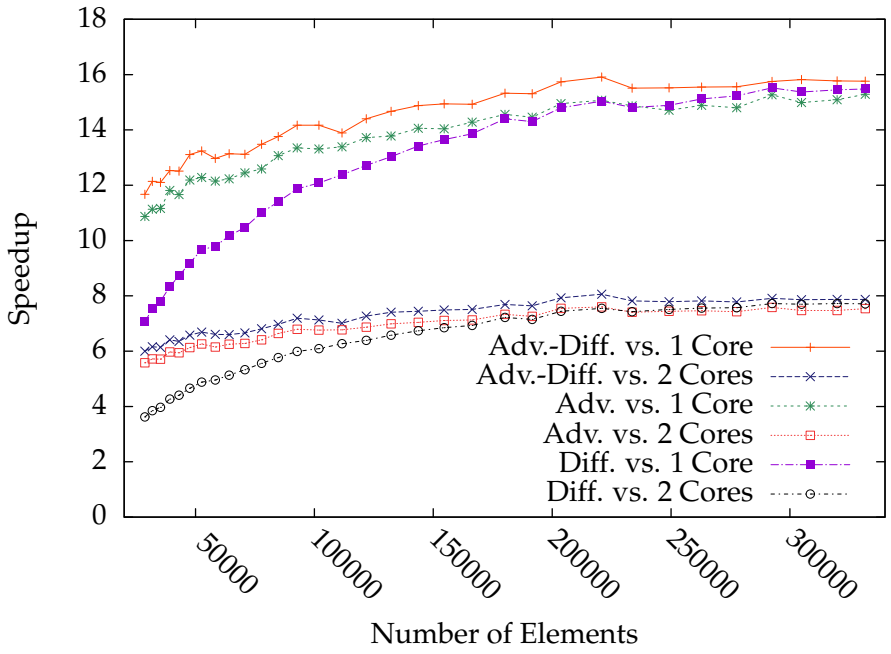


Figure 4.19: Speedup in the assembly phase from using the GPU for each problem over 1 and 2 CPU cores.

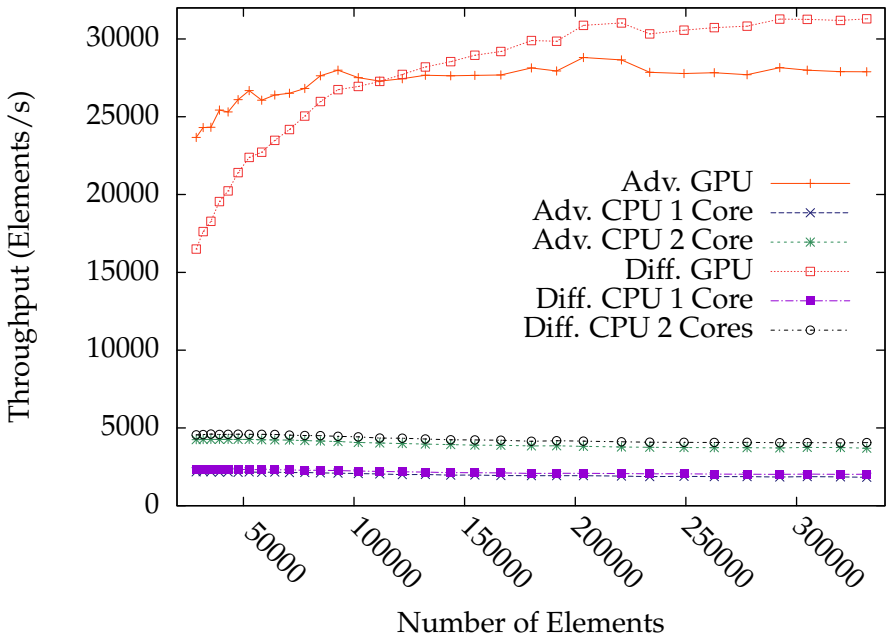


Figure 4.20: Throughput of assembly phase for each architecture and problem.

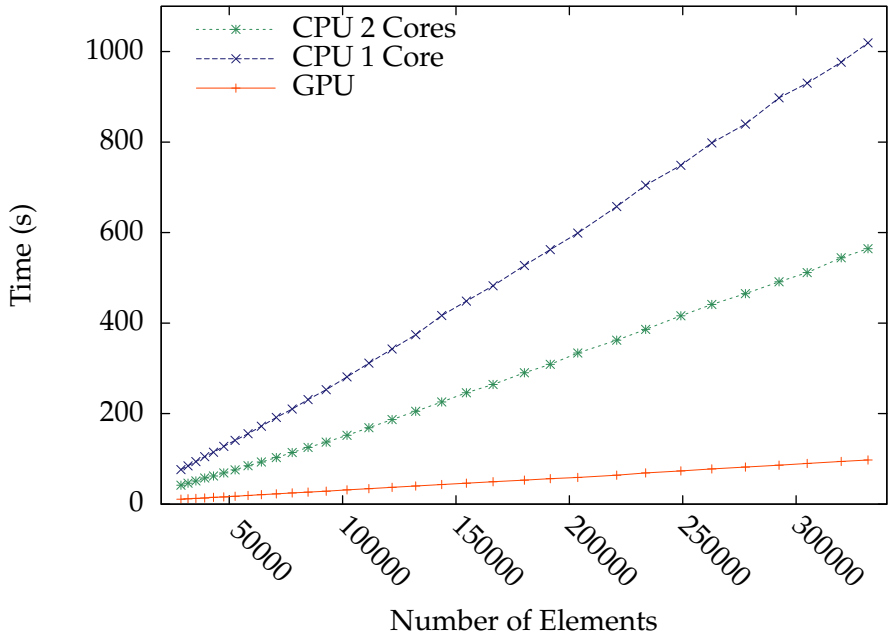


Figure 4.21: Total time running the Advection-Diffusion system for 200 timesteps.

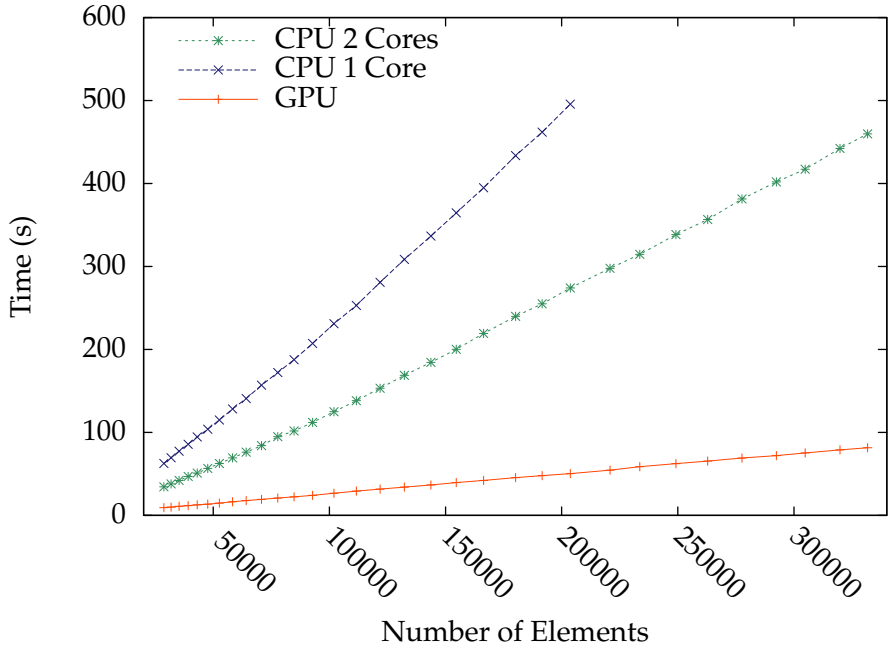


Figure 4.22: Total time running the Advection system for 200 timesteps. Results for the 1 CPU version for meshes larger than approximately 200000 elements are not shown, since the solver fails on these systems for unknown reasons.

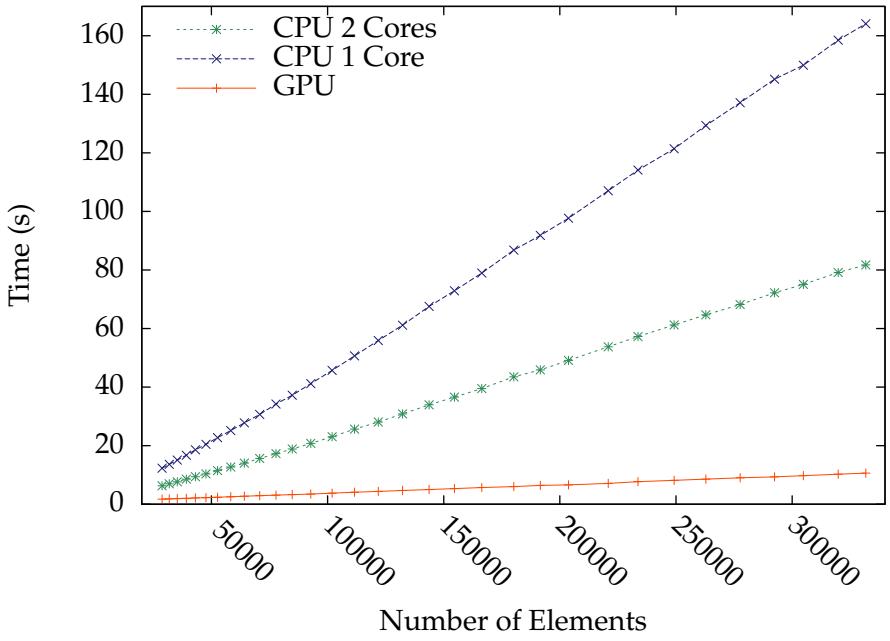


Figure 4.23: Total time running the Diffusion system for 200 timesteps.

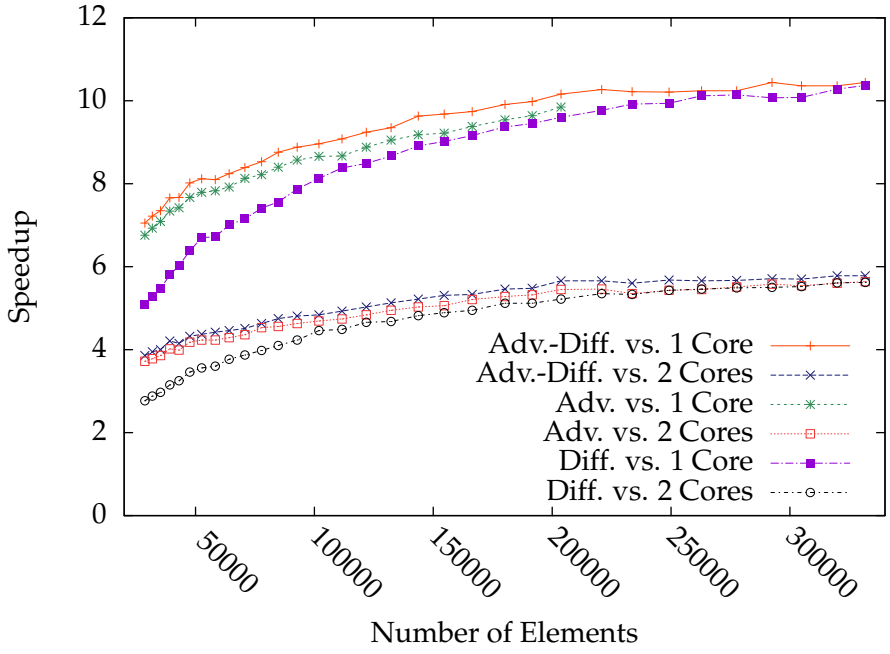


Figure 4.24: Overall speedup obtained using the GPU for each problem over 1 and 2 CPU cores.

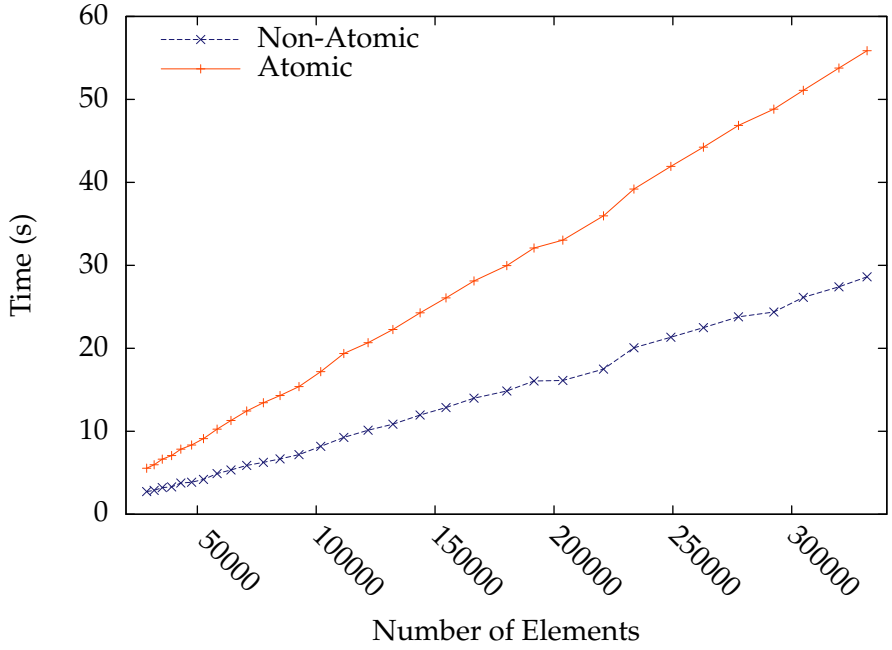


Figure 4.25: Times taken by the assembly phase for Advection-Diffusion on the GPU using atomic and non-atomic additions.

4.4.4 Performance Improvement

One area which is suspected to be the cause of relatively poor performance is the addition of values into the global matrix and vector. Since many threads add to these structures in parallel, atomic operations are required, which are expensive. In order to determine the overhead of these operations, we run the assembly phase of the advection-diffusion system again with these atomic operations replaced with non-atomic ones. Figure 4.25 shows a comparison of the time taken to assemble systems with varying numbers of elements.

It is clear that the atomic operations have a high overhead - we can see that for equivalent problems, the execution of the version using non-atomic operations is approximately twice as fast. It is not possible to use the non-atomic version without further modifications, since data races will occur and incorrect results will be summed into the matrix. However, if elements of the mesh were coloured such that no two elements of the same colour share a node, the non-atomic additions could be used since there will be no possibility for races. A colouring of a mesh may be computed offline, before the computation begins. This optimisation has been used to successfully increase the performance of the implementation described in [Komatitsch *et al.*, 2009] (see Section 3.2.5).

In order to determine how the GPU implementation may be improved so that it has higher performance, we also examine the performance of individual kernels. The CUDA Profiler was used to record profile data for the assembly phase of the advection-diffusion problem with a mesh consisting of 231562 elements. First, we examine the relative amount of time spent in each GPU kernel to determine which ones are more performance-critical. Figure 4.26 shows a plot of these times.

We can see that almost all of the GPU time is spent performing "addto" operations, where local matrices and vectors are summed into the global matrix and vector. Combined, these kernels occupy over 84% of the entire assembly phase, so are clearly the most performance critical. The `matrix_addto` and `matrix_addto_sum_diag` kernels occupy more execution time than `matrix_addto_diffusion` because they are used four times in the assembly phase. `matrix_addto_sum_diag` has less work to do than `matrix_addto`, as it only has to search the sparsity structure of the global matrix three times instead of the nine that `matrix_addto` requires. `vector_addto` is called five times in the assembly phase. Although it does not have to search a sparse matrix, it indirectly

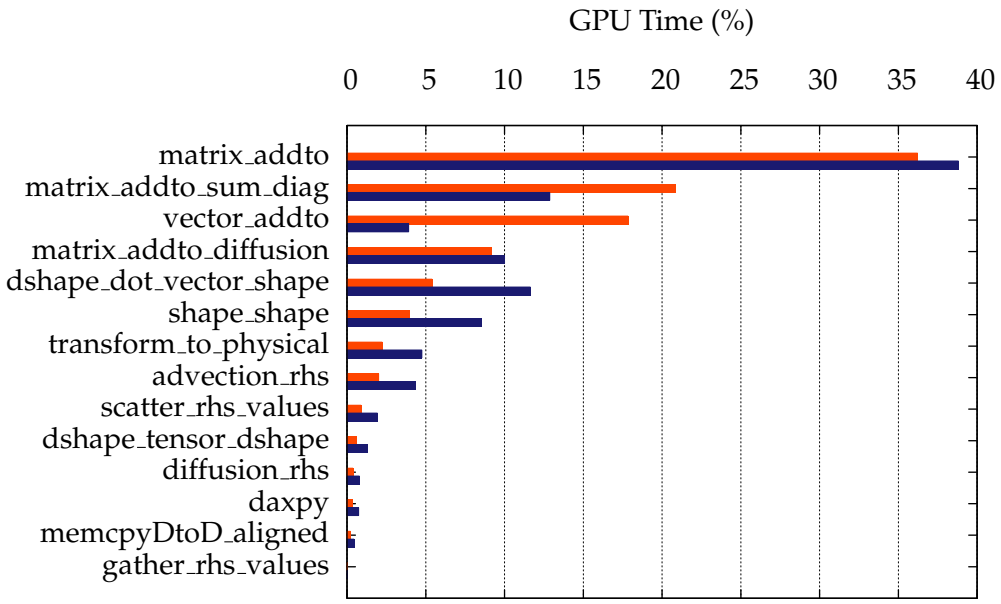


Figure 4.26: Relative time spent executing each kernel in the Advection-diffusion assembly phase. (Orange: with atomic operations. Blue: with non-atomic operations).

accesses elements of the global vector. This gives poor memory performance due to a lack of coalescing, and hence its execution takes a relatively large amount of time.

It is also of note that the atomic addto is partially responsible for the large proportion of execution times of these kernels, since they are the only kernels that contain this operation. Searching the sparsity structure of the matrix was originally thought to be the cause of the performance issues in these kernels, partially due to warp serialisation. However, the profiler also shows that only 20% of the branches diverge. This is probably due to the matrix containing relatively short rows (approximately 6.8 non-zeroes per row [Markall and Kelly, 2009]).

We see from Figure 4.26 that the proportion of time spent inside the non-addto kernels is doubled when non-atomic operations are used. We see a decrease in the spent executing the addto kernels, though not uniformly across all of them. We make the following observations:

- The proportion of time spent executing `matrix_addto` remains relatively unchanged between the atomic and non-atomic versions. This kernel now dominates the execution time since it has to search the sparsity structure of the matrix 9 times per element.
- The proportion of time spent executing `matrix_addto_sum_diag` is reduced by one third. Since it only searches the sparsity pattern three times per element, it is less dominant than the `matrix_addto` kernel.
- The proportion of time spent executing the `matrix_addto_diffusion` kernel is relatively unchanged, since it also requires 9 searches of the sparsity pattern per element.
- The proportion of time spent executing `vector_addto` is greatly reduced. Since the matrix is not accessed in this kernel, most of its execution time was previously spent performing the atomic addition.

We may also examine the usage of the available memory bandwidth of each kernel. Figure 4.27 shows the memory bandwidth utilisation of each kernel. The four addto kernels all show very poor memory bandwidth utilisation. Since there is very little computation performed in these kernels, this reinforces the notion that the atomic operations carry a heavy penalty, since the memory performance is limiting the speed of these kernels. We note that when non-atomic

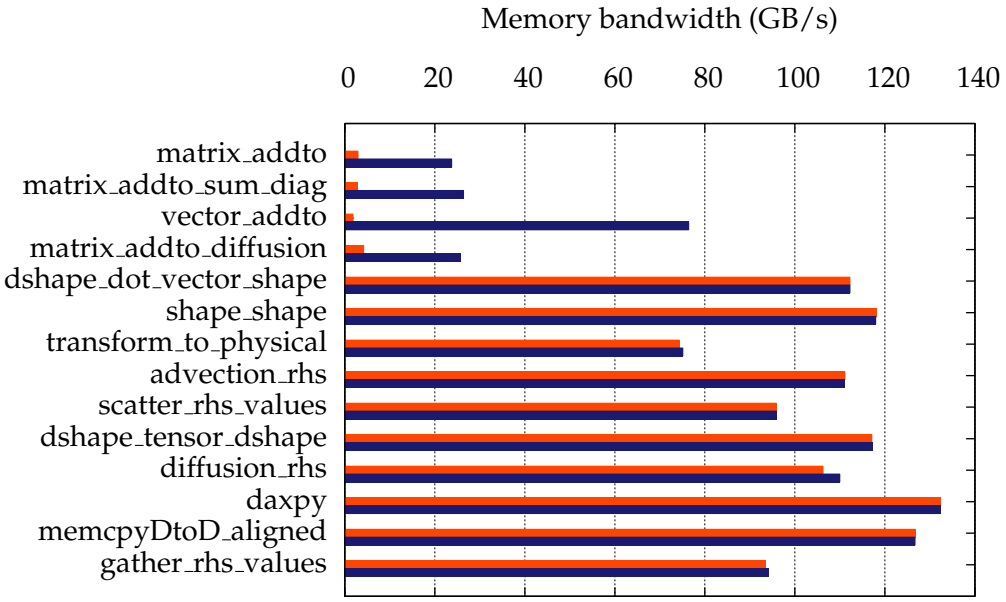


Figure 4.27: Memory throughput of each kernel in the Advection-diffusion assembly phase. (Orange: with atomic operations. Blue: with non-atomic operations).

operations are used, the memory bandwidth performance of all these kernels increases by an order of magnitude. However, their overall memory performance is still poor, due to lack of coalescing when searching the sparsity pattern of the matrix. The `vector_addto` kernel shows a much greater increase as it does not require the sparsity pattern to be searched.

The other kernels generally show good utilisation of the theoretical maximum memory bandwidth, suggesting that they are bandwidth bound. In order to increase the performance of these kernels, it will be necessary to increase their efficiency in using the available memory bandwidth. One method of increasing their utilisation of memory bandwidth involves fusing two or more kernels, and using shared memory or registers to pass the intermediate values between them instead of writing data back to arrays in global memory. However, increasing the size of kernels lowers occupancy, so the size of kernels must be traded off against their reduction in global memory usage. We see that the occupancy is relatively high for most kernels (Figure 4.28) in this implementation, which may permit fusions to increase performance. Exploration of the space of possibilities by hand is likely to be tedious and error prone - the ability to automate this optimisation process strengthens the case for the use of an automated tool that can explore this space.

An alternative method of decreasing the memory bandwidth requirements involves storing nodal data in a more efficient way. Currently, because data is extracted from the mesh for each node per element, there is some repetition of nodal data as each element is stored separately. This leads to the transfer of data for each node approximately 5-6 times per node, since each node belongs to 5-6 elements. A more efficient scheme involves partitioning the mesh into small chunks of adjacent elements, whose nodal data is small enough to fit in shared memory. At the beginning of a kernel, threads may cooperate to load a partition into shared memory, before the computation is performed for each element from the data in shared memory. This scheme will still lead to data being transferred multiple times for node boundaries, but decreases the total data transfer. Unfortunately the small size of the shared memory will limit the size of partitions, which will limit the effect that this optimisation may have.

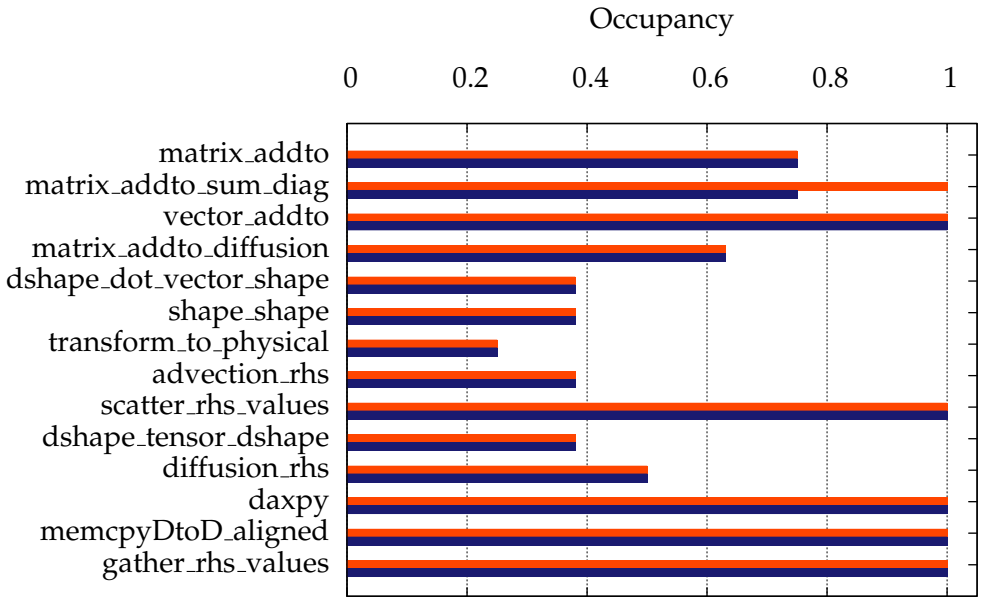


Figure 4.28: Occupancy of each kernel in the Advection-diffusion assembly phase. (Orange: with atomic operations. Blue: with non-atomic operations).

4.5 Conclusions

We have implemented a library of optimised CUDA kernels for finite element assembly, and made use of this library in the conversion of the assembly phase of the two test programs to a CUDA implementation[1]. Integration of these assembly phases with the Fortran source codes has been achieved. We have shown that the structure of the original Fortran source for the assembly phase is similar to that of the CUDA implementation, with the exception of the parallelism in the CUDA implementation.

Testing has shown that the CPU and CUDA implementations produce equal results up to an expected level of precision, and that the CUDA implementation yields a speedup of approximately 8 times over 2 CPU cores for the assembly phase in `test_advection_diffusion`. The overall speedup gained from using the GPU in `test_advection_diffusion` is approximately 6 times over the 2 CPU core implementation. Although we do not show such performance increases in the assembly phase of `test_laplacian`, this is not of concern since it is not highly representative of the finite element routines in Fluidity.

We have discussed optimisations that increase the performance of the assembly phase in CUDA, noting that ensuring coalesced memory access is particularly important. We have examined the performance of the CUDA implementation of the assembly phase of `test_advection_diffusion`, and shown that there are two main performance bottlenecks:

- The use of atomic operations for the addition of local matrices into the global matrix. This may be overcome by colouring the elements of the mesh such that no two elements of the same colour share a node, and performing additions into the global matrix for each colour in turn.
- The need to search the sparsity pattern of the global matrix when adding in terms from the local matrix. This issue may be overcome by using an alternative storage format.

The development of these manual translations facilitates the development of a UFL compiler by providing code that we use as a guide to what the compiler should output. We discuss the development of this compiler in the following chapter.

[1] See Appendix A for a summary of the kernels used in the assembly phase of each test program.

Chapter 5

A UFL Compiler for CUDA

5.1 Introduction

In this chapter we present a preliminary investigation into the implementation of a UFL compiler that outputs CUDA code. The development of this prototype compiler is based on the examination of the GPU-accelerated assembly routines for each of the test problems. This allowed development of the prototype compiler to be oriented towards generating one of these codes. As a result, code generated by the compiler makes use of the optimised kernel library described in Chapter 4.

The purpose of this investigation is not to provide a starting point for developing a fully-fledged compiler, but instead to discover issues that may be encountered during the design and implementation of such a compiler. As such, the resulting prototype compiler is limited in the range of UFL codes it can compile, but serves as a proof that generation of CUDA code from UFL specifications is feasible and practical.

5.2 Design

The design of the compiler is based around two main goals:

Completeness. We require the UFL compiler to be complete enough to generate at least one of the test codes.

Platform-Independence. Although we only intend to produce a compiler that outputs CUDA code, it should be designed such that modification to produce output for an alternative architecture is feasible. This is important since part of the motivation for using UFL is that it allows specification of a method uncoupled from any specific hardware implementation.

In order to meet these goals, the compiler is to be split into two parts. The *frontend* of the compiler will parse UFL and generate an *intermediate representation* (IR) of the code that is sufficiently expressive to represent the computation which must be performed. The *backend* will take its input from the frontend, and perform a translation from the IR to CUDA code. As development of the compiler was driven by generation of the code of the test problems, the design of the backend took place first. The design of the frontend followed, based on the information required by the backend.

5.2.1 The Backend

Examination of the CUDA code that implements each of the test problems reveals that they all have a similar basic structure, which consists of four functions, each performing a separate task:

Initialisation. Performs allocation of memory for all variables on the GPU, binds texture memory, and uploads the matrix sparsity pattern to the GPU.

Element Streaming. Performs copying of data for the Assembly phase from the host to the GPU.

Assembly. This function launches kernels to perform the required computations for the Assembly phase.

Finalisation. Performs deallocation of un-needed variables and un-binds texture memory.

In order to minimise the development effort for the prototype compiler, the CUDA code-generation backend was designed to be hardcoded to output four functions, each performing one of the operations described above. For the backend to know what code to generate in each of these functions, it requires lists of parameters to be passed to it that describe requirements in each of three categories:

Stream Variables. These are variables that are to be copied from the host to the GPU in the Element Streaming function. Their name, type, and their size as a multiple of the number of elements is required. An example of this type of variable is the list of node numbers of each element.

Internal Variables. An internal variable is one which stores data that is computed on the GPU and consumed by other GPU kernels only. The name, type, and size of each of these variables are also required. A example of an internal variable is the element-local matrix. Its value is generated by a kernel that evaluates a bilinear form, and is used as input to the kernel which adds values into the global matrix. Internal variables are specified separately to stream variables, since stream variables are copied over from the host in the Element Streaming function, whereas internal variables are only use by kernels in the Assembly function.

Kernels and Parameters. A list of the kernels to be launched and their parameters is also required. This list of kernels is translated into kernel calls in the Assembly function.

As well as the variables specified in the input, the backend adds some extra variables to these lists, which are always required. These include data structures that store the matrix and right-hand side vector being assembled, and code that copies the matrix sparsity pattern from the host is added.

5.2.2 A UFL Frontend

The list of variables and kernels, and their parameters forms the IR that is used to pass information from the frontend to the backend. The frontend must be capable of generating this list of variables and kernels from a UFL specification. The following sections detail how this is achieved.

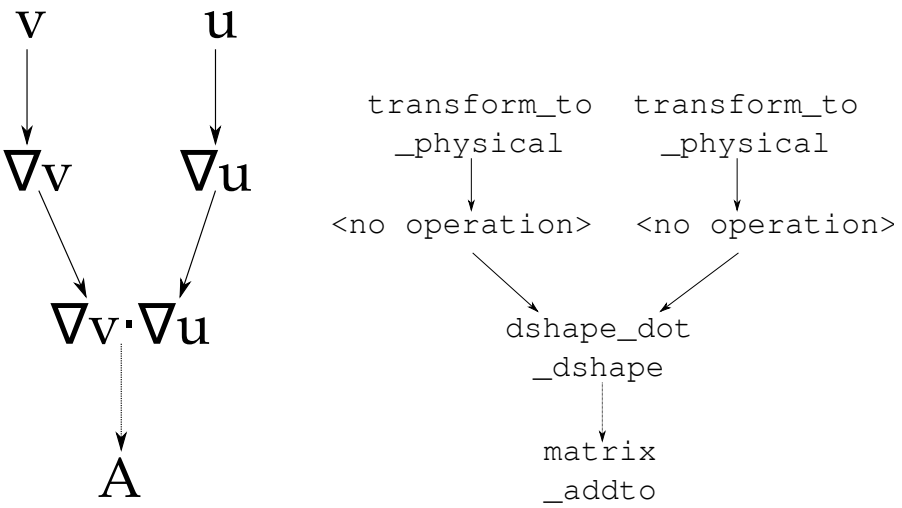
Directed Acyclic Graphs

A *Directed Acyclic Graph* (DAGs) is a data structure commonly used to represent expressions [Aho *et al.*, 2006, p.359]. Operations and operands are represented as nodes in the graph, and the dependences between nodes in the expression are represented by edges. Specifications of forms in the UFL Language may be represented as DAGs. We will consider an example UFL statement:

```
A=dot(grad(v),grad(u))*dx
```

The corresponding DAG for this expression is shown in Figure 5.1(a). In this DAG, the solid lines indicate those implied in the evaluation of the right-hand side, whereas the dotted line indicates a dependence as a result of the assignment of the right-hand side to `A`. We can see the edges in this graph represent the following dependences:

- In order to assign A, $\nabla v \cdot \nabla u$ must be computed.
- In order to compute $\nabla v \cdot \nabla u$, we must have access to ∇v and ∇u.



(a) Mathematical operation DAG.

(b) Kernel call DAG.

Figure 5.1: DAGs representing the UFL statement `A=dot(grad(v),grad(u)*dx`.

- In order to have access to ∇v and ∇u, we must have access to v and u.

The DAGs of mathematical operations may be converted to a list of kernels by performing a syntax-directed translation [Aho *et al.*, 2006, p.53], since there is roughly a 1:1 correspondence between nodes in the DAG and kernels that compute their result. One of several cases may be encountered at each node, and we deal with each one accordingly:

Test/Trial Functions. Encountering a node representing a test or trial function (such as v or u in Figure 5.1(a)), indicates that at some later node, the shape functions or their derivatives, and the Jacobian for the transformation to physical space will be required. Calling the `transform_to_physical` kernel ensures that these requirements will be satisfied.

Gradients of Test/Trial Functions. A test or trial function will have appeared earlier in the DAG and forced a call to `transform_to_physical`, producing the transformed derivatives of the shape function for the test or trial function. Therefore, we take no action when this type of node is encountered.

Functions. Functions that are not test or trial functions have a scalar value dependent on their position in the mesh. When encountering a node of this type, we need take no action since the field will have been extracted from `state` by an earlier call to `scalar_fields` or `vector_fields` (see Section 5.2.3).

Dot Products. The action taken when a dot product is encountered depends on the operands of the dot product. In the assembly routines we have considered, the integrand may take one of four possible forms (where a and b may be test or trial functions, c is a function and μ a tensor):

- $a \cdot b$: In this case we must invoke the kernel `shape_shape` on the operands
- $\nabla a \cdot \nabla b$: In this case we must invoke the `dshape_dot_dshape` kernel.
- $\nabla a \cdot cb$: In this case we must invoke the `dshape_dot_vector_shape` kernel.
- $\nabla a \cdot \mu \cdot \nabla b$: In this case we must invoke the `dshape_tensor_dshape` kernel.

Conversion of the DAG of mathematical operations to one of kernel calls is accomplished by visiting each node in the graph and swapping mathematical operations for kernel calls using the

rules outlined above. The final stage of the frontend will consist of converting this graph of kernels to a list of kernels suitable for passing on to the backend. The complete algorithm is detailed in Algorithm 1.

Algorithm 1: Generation of a list of kernel calls starting from an integrand.

Input: An integrand, I
DAG = Graph(I) ;
KernelList = An Empty List ;
for *Each node N in DAG* **do**
  if *N a test or trial function* **then**
    Name = Field name of N ;
    Append `transform_to_physical(` Name`)` to *KernelList* ;
  else if *N is a dot product* **then**
    O_1 = First operand of N ;
    O_2 = Second operand of N ;
    if O_1 *and* O_2 *are Gradients of test/trial functions* **then**
      Append `dshape_dot_dshape(`O_1,O_2`)` to *KernelList* ; `/* Returns Result */`
    else if O_1 *and* O_2 *are test/trial functions* **then**
      Append `shape_shape(`O_1,O_2`)` to *KernelList* ; `/* Returns Result */`
    ...; `/* Two further cases omitted */`
    Append `csr_addto(`*Result*`)` to *KernelList*;
  else if *N is a product* **then**
    Name = Function name associated with N ;
    Append *Name* to *KernelList* ; `/* Returns Result */`
    Append `scalar_field_vaddto(`*Result*`)` to *KernelList* ;
  Return Depth_First_Ordering(*KernelList*)

Variable Name Generation

As well as providing a list of kernels to the backend, the frontend must produce a list of variable names as described above. These names must be generated such that they are unique, and kernel parameters must reference the variable names such that they operate on the correct data. In order to ensure the uniqueness of variable names and their correctness as kernel parameters, we the following conventions for the generation of names:

- For internal variables, we base their name on the name of the field being operated on and a description of the type of output produced by the kernel. For example, since the `dshape_dot_dshape` kernel produces an output that is a local matrix, the form of the output variable name is `lmat_<op1_fname>_<op2_fname>` where `opX_fname` is the name of the field associated with the operand X.

- Stream variables are named after the data they represent and the field they come from. For example, from a field named `Tracer`, a variable storing the shape functions would be called `Tracer_n`, and the derivatives of the shape functions would be called `Tracer_dn`. This follows the convention in Fluidity, in which `n` and `dn` generally represent a shape function and its derivatives.

Variable names are generated as each node in the graph is visited. To ensure that all variable names are passed to the backend, a lists of the Stream and Internal variables are kept, with each new variable added to the appropriate list as it is generated.

5.2.3 Integration With Fluidity

All of the GPU-accelerated assembly routines in the test problem make use of an external Fortran module which acts as an intermediary between the data structures defined by the Fluidity API and the data structures used in the GPU implementation. Code generation of this intermediate module would be likely to be relatively straightforward, since its implementation in each case consists of a loop over the elements in the mesh, which copies data from the Fluidity mesh data structures into a separate location using the alternative storage layout described in Section 4.2.8. Because of the trivial nature of the intermediate Fortran modules, it is unnecessary to produce a proof-of-concept code generator for these portions of code.

A more challenging problem is that of how we allow the programmer to specify which Fluidity data structures to access in the UFL codes. UFL itself does not provide any method for access to external data. It is therefore necessary to extend to UFL specification to make it fit for this purpose.

The Fluidity API provides a *state dictionary*, which allows for the access of any data structure that is part of the simulation status to be accessed by name. This is fortunate, as it allows us to extend the UFL specification in such a way that the user specifies fields that are the source of their data by name. For example, the user may wish to access a scalar field named `psi` (as used in `test_laplacian`). We may support this action, and similar actions for other types of fields by providing a notation as in the following example:

```
psi=state.scalar_fields("psi")            /* Extract a scalar field */
coord=state.vector_fields("coordinate")   /* Extract a vector field */
diffusivity=state.tensor_fields("mu")     /* Extract a tensor field */
```

This now allows the user to perform the usual UFL operations on the variable `psi`, such as:

```
v=TestFunction(psi)
u=TrialFunction(psi)
f=Function(psi)
```

This extension proves to be sufficient for access of all the data structures required in the test problems. Although there are other types of data pertaining to the state of the simulation in existence, these are only used in more complex simulations. The notation `state.<x>` is expected to be straightforward to extend to provide support for accessing these other data types.

5.3 Implementation

5.3.1 The Python Frontend

The FEniCS project provides an implementation of the UFL language written in Python. This provides a good starting point for implementing a UFL compiler. Using this distribution of UFL, it becomes possible to execute a UFL specification in the Python interpreter, leading to the construction of data structures representing the UFL script. This allows us to avoid writing a parser for UFL.

The UFL distribution provides algorithms for working with these data structures, in particular `algorithms.graph`. We make use of this package in the following ways to deal with representations of bilinear forms:

- We use the class `algorithms.graph.Graph` to convert the terms in a form to a DAG representing the integrand.
- Subsequently, we use `algorithms.graph.depth_first_ordering()` to serialise the DAG.
- Each different mathematical operator is represented by a different class. For example, ∇ is represented by the class `Grad`. When we visit each node in the serialised DAG, we test to see

if it is an instance of one of the classes representing the operators we are testing for. Once we have determined the class of the node, we can easily infer which kernel to add to the kernel schedule list.

Since we intend to use the UFL distribution, which requires a UFL script to be executed to use it as input, we needed to find a way to force the execution of a UFL script to result in code generation. In order to achieve this, we needed to find suitable method which would be present in each of the UFL scripts for the test problems that we could define such that it calls the code generator. Examination of the UFL for each test problem reveals that they all end with a `solve` statement, therefore this is the natural choice for such a function. The syntax of the function is `solve(x, A, b)` where `A` is a form specifying how the matrix is assembled, `b` is a form specifying how the right-hand side vector is assembled, and `x` will store the result.

The implementation of the `solve` function is as shown in Algorithm 2. The function `GenerateKernels` is an implementation of Algorithm 1, which also creates lists of variable names as it traverses the DAG. Since the algorithm will create several duplicates of some kernels, and of the variables, it is necessary to remove these duplicates before calling the backend. Although duplication of the kernels will not cause the generated code to produce incorrect answers, it will result in more computation than is necessary.

Algorithm 2: Frontend portion of the code generator (built into `solve` function).

Input: LHS A, Unknown vector $\mathbf{x}$, Known vector $\mathbf{b}$
$LHSInt \leftarrow$ Integrand in A ;
$[LHSKernels, LHSInternalVars, LHSStreamVars] \leftarrow$ GenerateKernels($LHSInt$) ;
$RHSInt \leftarrow$ Integrand in A ;
$[RHSKernels, RHSInternalVars, RHSStreamVars] \leftarrow$ GenerateKernels($RHSInt$) ;
$Kernels \leftarrow LHSKernelVars + RHSKernelVars$;
$InternalVars \leftarrow LHSInternalVars + RHSInternalVars$;
$StreamVars \leftarrow LHSStreamVars + RHSStreamVars$;
Uniqify($Kernels$) ;
Uniqify($InternalVars$) ;
Uniqify($StreamVars$) ;
Backend($Kernels, InternalVars, StreamVars$) ;

Implementation of the `state` class for access to fields within Fluidity was achieved by defining its methods `scalar_fields` and `vector_fields` such that they return an object of type `Element`. This object has an extra attribute, `name`, which stores the name of the field. When the kernel and variable generation algorithms visit nodes in the DAG, this field is preserved, and is used to find the field name that the node refers to.

The choice to return an `Element` object was driven by the requirements of the constructor for the classes `TestFunction` and `TrialFunction`, and similar classes. Normally a UFL user would initialise one of these functions using code similar to the following:

```
ele=Element("Lagrange", "triangle", 1)
v=TestFunction(ele)
```

The choice to return an `Element` type satisfies all of the constructors for these functions. The type of element (triangular elements with degree 1 Lagrange polynomials as basis functions) does not matter in our implementation, since these attributes of the `Element` are discarded when the DAG of mathematical operations is converted to a DAG of kernels.

5.3.2 The Code Generation Backend

The backend is implemented using the ROSE Compiler Infrastructure [Quinlan *et al.*, 2009a]. ROSE is intended as a toolkit for building source-to-source translation tools. However, it is possible to

build an *Abstract Syntax Tree* (AST) [Aho *et al.*, 2006, p.41] representing a program in ROSE's IR, SAGE III. The interface provided by ROSE for building an AST is relatively straightforward; one may construct objects representing AST nodes, and use the provided methods to insert them at specified points in the AST. For a full description of the ROSE APIs, see [Quinlan *et al.*, 2009b].

The backend represents each variable using instances of a class `Variable`, and each kernel launch using a class `Kernel`. Lists of Internal and Stream variables, and the list of kernel launches are represented using standard STL data structures. The backend expects its input to be passed in the form of these data structures, necessitating an interface layer between the frontend and the backend. The *Simplified Wrapper and Interface Generator* (SWIG) [Beazley, 1996] was used to create a Python interface to the required classes and the backend, in the form of a Python module. This Python module is loaded by the frontend and is also used to call the backend. The implementation of the backend operates in the following way:

- A new AST with an empty source file is created. Each of the variable lists are iterated over, and for each variable, a global variable declaration is created and inserted into this source file in the AST. The list of kernels is referred to in order to generate forward declarations for each of the kernels that are to be called. Each kernel declaration needs to be prepended by the keyword `__global__`, which is not part of standard C or C++ supported by ROSE. In order to get around this, ROSE's mechanism for inserting arbitrary strings at points in the AST was used to insert the extra keyword before each declaration.
- The Element Streaming function is created and inserted into the AST, which receives as many parameters as there are stream variables. For each Stream variable, a call to `CudaMemcpy` is inserted into the function, to copy the data from the host to the GPU.
- When generating the Assembly function, the list of kernel calls is iterated over to determine which kernels need to be called. Since CUDA kernels are launched using the syntax `kernel_name<<<BlockDim,GridDim>>>(...)`, it was necessary to append the string in triple chevrons to the name of each kernel before it is inserted into the AST. Although this is technically not legal C++, the generated code is syntactically-correct CUDA code.
- To generate the Initialisation function, the backend iterates over each of the lists of variables, and inserts a `cudaMalloc` statement for each one. This ensures that space for each variable is allocated before it is used. Additionally, calls to `cudaMemcpy` copy the matrix sparsity pattern over to the GPU, and calls to `cudaBindTexture` bind it to texture memory.
- Generation of the Finalisation function is partly similar to the Initialisation function. For each of the Internal and Stream variables, a call to `cudaFree` is inserted. Calls to un-bind the matrix sparsity pattern (using `cudaUnBindTexture`) are inserted, but the matrix is not freed using `cudaFree`, since it will be required by the solver phase.
- Finally, two `#include` statements are added to include code stored in external files, which does not change. These are `gpu_declarations.h`, which determines the size of thread blocks and the grid, and `gpu_static.h`, which contains functions to add the boundary contribution from the CPU to the GPU right-hand side vector.

After the construction of the AST, the ROSE "Unparser" is called. The Unparser traverses the AST and writes the output code to disk. As this implementation of the compiler is a prototype, the output is coded to be written to a file called `gpu_assemble.cu`, since a choice of output filename is not important.

5.4 Testing

5.4.1 Generation of Test Input for the Backend

It is possible to perform testing of the backend by writing a standalone C++ program that generates STL data structures containing the Internal and Stream variables, and a list of kernel launches

and their parameters. For example, the code shown in Figure 5.2 specifies a particular invocation of the `matrix_addto` kernel. The parameters `val` and `size_val` are part of the data structures representing the matrix, `ele_psi` is the node numbers of each element of the field `psi`, `lmat` is an element-local matrix, and `n` is the number of elements. In order to test the backend, similar constructs to the one shown above were used to create a test input to the backend which was expected to cause it to output a code equivalent to the implementation of `test_laplacian` described in Section 4.2. The output code was tested to verify that it ran correctly and produced the same results as the hand-translation of `test_laplacian`.

```
stringList *params = new stringList();
(*params).push_back(string("val"));
(*params).push_back(string("size_val"));
(*params).push_back(string("ele_psi"));
(*params).push_back(string("lmat"));
(*params).push_back(string("n"));
launchList.push_back(kernelLaunch("matrix_addto",params));
```

Figure 5.2: Code to construct an invocation of the `matrix_addto` kernel in the IR.

5.4.2 Generation of Test Input for the Frontend

In order to test the frontend, a UFL implementation equivalent to the assembly phase of the `test_laplacian` test program was written. The code for this implementation is shown in Figure 5.3

```
P = state.scalar_fields("psi")
v=TestFunction(P)
u=TrialFunction(P)
f=Function(P)
f.name="shape_rhs"
A = dot(grad(v),grad(u))*dx
RHS = v*f*dx
solve(P, A, RHS)
```

Figure 5.3: UFL Code equivalent to the assembly phase in `test_laplacian`, used for testing the frontend.

This code was executed using the frontend, which caused the code generator to be invoked. The resulting output code was again equivalent to the hand-translation of `test_laplacian` and produced the same results, showing that the implementation of the UFL compiler functions correctly in this case.

5.4.3 Generation of Further Inputs

In order to further test the UFL compiler, we may consider producing a UFL script to solve the Helmholtz Equation. The Helmholtz Equation is:

$$\nabla^2 u - \lambda u = f \tag{5.1}$$

This is clearly similar to the equation solved by `test_laplacian` , with the addition of an extra term. Solving of this equation using the finite element method is also very similar, which is achieved by assembling and solving a discretisation of the following equation:

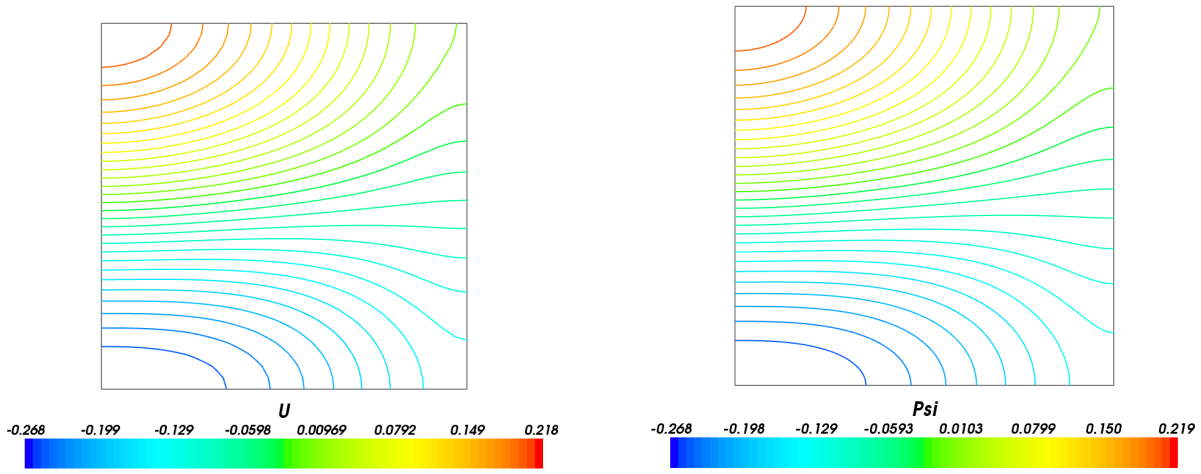
$$\int_\Omega \nabla v \cdot \nabla u \mathrm{d}X + \lambda \int_\Omega vu \mathrm{d}X = -\int_\Omega vf + \int_{\partial\Omega} v\nabla u \mathrm{d}s \tag{5.2}$$

In order to change the UFL implementation of `test_laplacian` shown above to solve this equation (when $\lambda = 20$), a change to the assignment to `A` is necessary:

```
a = (dot(grad(v), grad(u))+(20)*dot(v,u))*dx
```

The modification of this term results in a larger and more complex DAG. Due to time constraints, it was not possible within the project to add enough extra logic to the kernel scheduling algorithm of the frontend to successfully compile this UFL code using the frontend. However, it is possible to emulate the desired behaviour of the frontend. This is achieved by modifying the C++ test code described above to add extra kernel calls to `shape_shape` and `matrix_addto`, which causes the extra term to be assembled into the matrix.

To test the correctness of this generated code, code implementing the same problem was generated using FEniCS Dolfin to provide a reference output. The solutions to the equation produced by FEniCS Dolfin and the generated code are shown in Figure 5.4. It can be seen from these two plots that the solutions are very similar. The slight discrepancy, which can be seen as a difference of 0.001 in the maximum value on the legend is due to the coarseness of the mesh used in the FEniCS implementation. However, this result shows that the code generator has generated correct code to solve the Helmholtz equation.



(a) Generated by FEniCS.

(b) Generated by the UFL Compiler output.

Figure 5.4: Visualisation of solutions to the Helmholtz Equation with $\lambda = 20$.

5.5 Conclusion

We have discussed the design and implementation of a UFL compiler that outputs CUDA code. The design has been closely guided by the manual implementations discussed in the Chapter 4. In order that the compiler may generate code for different target architectures, it is composed of a frontend, which inputs UFL code and produces an intermediate representation, and a backend which generates code from this intermediate representation.

The compiler has been tested with a UFL specification of the `test_laplacian` program. The output from the compiler was found to be equivalent to the manual implementation of the assembly phase in `test_laplacian`. The backend was also tested with an input which caused it to generate code to solve the Helmholtz equation. The solution produced by this code and an implementation of the same problem in FEniCS Dolfin were found to be equivalent.

Although the compiler only supports a limited subset of UFL, it has been shown that the generation of CUDA code from high-level UFL specifications is feasible. The addition of support

for other parts of the language, which will allow the compiler to compile UFL code representing the assembly phase of `test_advection_diffusion`[1], is left for future work.

[1] See Appendix B for the UFL code representing the assembly phase in `test_advection_diffusion`

Chapter 6

Evaluation

6.1 Introduction

In this chapter we seek to evaluate the work of the project. We compare the implementation described in Chapter 4 to the others discussed in Chapter 3, and discuss how the UFL compiler demonstrates the feasibility of generating CUDA code from UFL specifications. We also explore whether continued research in the direction of this project is worthwhile and feasible.

6.2 Examination of the Implementations of the Assembly Phase

We may regard the experimental implementations of the assembly phase as being successful in showing that the use of GPUs and CUDA in particular leads to improvements in performance in finite element assembly. We have seen in Section 4.4 that the implementation increases the speed of the assembly phase by approximately 8 times. Although this speedup of almost an order of magnitude has been obtained, it is clear that more could be done to improve the performance of the CUDA implementation. We briefly highlight optimisations that might be used improve the performance of our implementation which have been reported in other work:

Fusion of Kernels. In Section 3.2.2, optimisation of an implementation is discussed in which kernels implementing different operations are fused. We have also seen that the majority of kernels in the implementation are limited by memory bandwidth (Section 4.4.4) due to the storage of intermediate results in shared memory. It can therefore be concluded that we could further improve performance by implementing kernel fusion optimisations.

Mesh Partitioning. In Section 3.2.3, dividing the mesh into partitions small enough to fit in to shared memory is discussed. Since node data is loaded once per element in our implementation, loading partitions of elements into shared memory will decrease memory bandwidth pressure since nodes interior to to the partition will only need to be transferred once.

Packing of Elements. The storage layout described in Section 3.2.3 and pictured in Figure 3.2 allows elements in a partition to be efficiently loaded into shared memory. In our implementation, the layout of element data is such that the data for every element at a given node is consecutive in memory. This data layout will require a gather operation if a partitioning scheme is used, since data for a single element is not contiguous. In order to implement a partitioning scheme, we must also use a data layout for element data similar to the one shown in Figure 3.2.

Element Colouring. In Section 3.2.5, a colouring of elements such that no two elements of the same colour share a node is described. Our performance analysis (Section 4.4.4) shows that the atomic operations have a very high cost. Use of this colouring and assembly of each colour in turn will eliminate the need for these atomic operations, bringing an increase in performance.

Matrix Storage. Although none of the implementations discussed in Chapter 3 mention the matrix storage format, the choice of alternative storage formats should be evaluated. It is shown in Section 4.4.4 that the most expensive operation once atomic operations have been eliminated is the need to search the sparsity pattern of the global matrix, which is stored in CSR format. An alternative storage format that requires less operations to find the location of a particular element of the matrix may be more efficient on the GPU.

6.3 Discussion of the UFL Compiler

We have seen in Section 5.4 that the implementation of the UFL compiler is able to produce output equivalent to the manual implementation of the assembly phase for `test_laplacian`. The implementation is limited due to time constraints within the project, and there are no technical barriers to supporting a larger subset of UFL so that the compiler may be used to generate code for the `test_advection_diffusion` assembly routine.

It is questionable whether the most pragmatic design has been used for architecture of the compiler. Since the frontend consists of a mechanism for both reading in a UFL file and producing a list of kernel calls that are to be generated, this limits the flexibility of the backend. For example, the list of kernels that make up a CUDA implementation may differ greatly from the list of kernels that would be used to make up a Cell implementation of the same assembly process. This is likely to make the implementation of backends for a diverse set of architectures difficult.

Defining the intermediate representation to be more similar to the DAGs produced by the `ufl.algorithms` package may be considered a better choice than defining it as a list of kernel calls. This will allow each backend to perform its own target-specific manipulations of the DAG before choosing which kernels should be used for the implementation. Additionally, since the DAG preserves the mathematical operators and their operands, this allows the backend to generate specialised kernels that implement any operator or combination of operators specified in the DAG.

A deviation from the UFL specification which the compiler implements involves the specification of a function. In UFL, the right-hand side function used in `test_laplacian` (Equation 2.9) is specified using the following declaration:

```
f=Function(P, "(2*1.25*pi*cos(0.5*pi*x[0]))*cos(pi*x[1])+0.5*pi*sin(0.5*pi*x[0])")
```

However, the implementation of this function by the compiler would require the generation of a specialised kernel that evaluates this function over the domain. Alternatively, it may be composed of a large number of generic kernels that perform basic mathematical operations, which would be quite inefficient and would require substantial manual work to implement in the compiler and kernel library. In order to overcome this issue, the following code is used instead:

```
f=Function(T)
f.name="shape_rhs"
```

In this code we specify that in order to evaluate the function `f`, we should call the kernel `shape_rhs`, which has been written in CUDA by the programmer. This clearly causes an issue since it breaks the abstraction from the implementation which UFL provides. However, since it is sometimes difficult to express more general computations in UFL, including the facility to call an arbitrary kernel may allow a programmer additional flexibility where necessary. We stress that the ability to define a function in UFL without having to resort to calling a kernel should be included in the UFL compiler as part of future work.

Finally, we note that the UFL compiler does not generate Fortran source necessary to integrate the CUDA assembly routine with a program written using the Fluidity APIs. The present implementation requires the programmer to write code which extracts the necessary data from the mesh data structures, perform a transposition of this data, and call the CUDA assembly functions. As we have outlined in Section 5.2.3, there is enough information present in the UFL specifications to generate a Fortran module that implements these operations. The development of an addition to the backend that outputs code to perform these tasks is left for future work.

6.4 Examination of UFL

In deciding whether and how to continue the work that has been completed throughout this project, it is necessary to question whether the UFL is an appropriate choice of language upon which to base our work. We argue that UFL provides an appropriate starting point for the development of tools for generation of finite element assembly code, but the language must eventually be extended with additional facilities. We support this claim with the following arguments:

- It is straightforward to express finite element methods using UFL, as the notation it provides is very close to that of the mathematical formulation of a finite element method. For example, we have seen in Section 5.4 that the assembly and solution of the system solved by `test_laplacian` may be specified using less than 10 lines of UFL, which have a close correspondence to the variational form of the problem given in Equation 2.11.

- Since UFL does not allow the programmer to give any specification regarding the implementation, the same source may be compiled to multiple target architectures without modification. This also permits automated exploration of the optimisation spaces, using techniques analogous to those used by the Tensor Contraction Engine to optimise the evaluation of tensor expressions for a particular target (Section 3.3). Additionally, the exploration of these optimisations using the compiler does not break the abstraction provided by UFL.

- UFL is a versatile language for the specification of finite element methods, allowing any problem that can be specified in the weak form to be described, using a variety of element types (although we have focused on order 1 piecewise continuous polynomial elements in this project), including those for DG methods. Since the Finite Volume and Finite Difference methods may also be formulated in terms of the finite element method with an appropriate choice of basis functions [Sherwin *et al.*, 2009, p.2-5], we claim that UFL might be also be used to specify these types of method.

- In spite of its versatility in the aspects described above, the UFL language does not provide support for describing iteration in a method. In order to overcome this, we must either ensure that iteration is specified externally to a UFL specification, or we must augment the language with constructs to specify iteration. The first approach is likely to break the abstraction provided by UFL. For example, if the description of a single iteration were captured using UFL, and iteration expressed in program written in a low-level language that makes calls to the generated code, unnecessary data transfer will be difficult to avoid. Consider an example loop written in Fortran:

  ```
  do timestep=1,FINALTIME
    call advection_diffusion_step(solution(timestep), solution(timestep+1))
  end do
  ```

  In this example the `advection_diffusion_step` function computes the solution at time $n+1$ given the solution at time n, where `solution` is a vector storing the solution at each point at each time. It is highly likely that unnecessary data transfer between the GPU and the host will occur at every timestep, which is likely to be unnecessary if the solution at every timestep is not desired. In order to prevent this occurring, a complicated data-flow analysis will be required to prevent the unnecessary transfers. Furthermore, determining at which point data should be transferred may not be possible without additional knowledge unavailable to the compiler.

  Therefore, it is clear that the addition of constructs for iteration is necessary. The design of these constructs is left for future work, since they must be carefully chosen so that they do not break the abstractions already in place, but do not preclude flexibility in the compiler's choice of optimisations, and the generation of code for a variety of targets.

6.5 Summary

We have discussed the manual implementations of the assembly phases, and we conclude that although considerable speedups have been obtained, we may further improve our performance results by using additional optimisations. We have concluded that the UFL compiler demonstrates the feasibility of generating CUDA code from UFL, although its design requires modification for future work. Finally, we have shown that UFL is a viable platform for further developments in the area of this project.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

We have demonstrated the following contributions throughout this report:

- We have surveyed and evaluated existing approaches to automatically generating finite element assembly code, and discussed the state of the art in GPU acceleration of finite element assembly (Chapter 3).
- We have described the implementation of a library of kernels used in the assembly phase on the NVidia Tesla architecture. Strategies that have been used to optimise performance have been discussed (Chapter 4).
- We have shown how this library is used in the assembly phase for a variety of test problems, resulting in performance improvements of almost an order of magnitude over the equivalent CPU implementation on typical hardware (Chapter 4).
- We have described a prototype implementation of a UFL compiler which outputs CUDA code, and how the conversion from UFL to CUDA is performed (Chapter 5).
- We have evaluated the implementation of the UFL Compiler and shown that the construction of a flexible UFL compiler which supports multiple backends and optimisations is feasible and worthwhile (Section 6.3).
- We have shown that the Unified Form Language is a viable platform for further research in this area (Section 6.4).

7.2 Further Work

7.2.1 Performance Optimisation of the GPU Implementations

We may attempt to further optimise the GPU implementations described in Chapter 4 using the techniques outlined in Section 6.2. These optimisations include kernel fusions, mesh partitioning, element packing and padding, element colouring, and the investigation of alternative matrix storage formats.

7.2.2 Completion of Support for UFL

The compiler described in Chapter 5 supports a fairly limited subset of the UFL specification at present. Increasing its coverage of the specification will allow the generation of a wider variety of finite element assembly routines. It is of interest to attempt to compile the UFL for the assembly phase of `test_advection_diffusion` and make comparisons with the hand-written code.

7.2.3 Implementation of Additional Backends

The prototype UFL Compiler presently only supports a CUDA target. Other target architectures, such as the Cell processor or multicore CPUs may be supported by the development of new backends. Before these alternative backends may be implemented, the intermediate representation requires modification so that it more effectively communicates information about the mathematical operations specified in a UFL script from the frontend to the backend.

7.2.4 Generation of GPU Kernels

At present, the library of kernels described in Chapter 4 is used as the basis for generated assembly routines. This limitation requires a programmer to develop new kernels to implement operations not supported by the library. In order to overcome this limitation, the addition of the capability to generate kernels based on a mathematical specification needs to be developed. This will also allow the specification of functions to be given in UFL rather than in a low-level language such as CUDA (Section 6.3).

7.2.5 Automated Exploration of Optimisations

As we have seen in Sections 3.3 and 2.5, the UFL compiler may make decisions about how any given specification of a method is implemented. We may extend the implementation of the compiler so that it applies different transformations to the generated code in order to discover an optimal implementation. For example, the CUDA backend may be used to perform fusion of various kernels. Alternatively, a backend targeting a multicore processor may be able to make use of SSE intructions to improve performance. Combined with the functionality to generate kernels, the compiler may also be used to explore other transformations such as loop unrolling or constant folding.

7.2.6 Development of Interface Code

At present, a programmer has to write a Fortran module which extracts data from the mesh and calls the code generated by the UFL Compiler. The compiler may be augmented with the capability to automatically generate a Fortran module for this purpose. This will allow a programmer to integrate generated code with Fluidity in a much more straightforward manner; they need only to include a `use` statement referencing the generated code, and pass the state dictionary of Fluidity to the generated routine when required. This facility will greatly ease the integration of the generated code with the Fluidity codebase.

7.2.7 Capture of Iteration in UFL

As described in Section 6.4, it is not possible to express iteration using UFL. It is necessary to examine how iteration might be captured before extending the UFL specification, as the best strategy for doing so is non-obvious.

7.3 Manifesto

It is intended that the research conducted throughout this project is to be used as a platform from which to work towards the integration of high level specifications into the Fluidity codebase. This integration will permit the use of generated code for solving complex fluid dynamics simulations involving multiple phases (water, air, solids etc.) and and fields (temperature, pressure, density, salinity, etc.). The high-level specifications will be used to generate highly-optimised target-specific code, resulting in performance greater than may be obtained by human effort. The high level of abstraction will enable the frontiers of the finite element method to be explored by facilitating the coupling of finite element simulations with those from other domains.

Appendix A

Optimised Kernel Library

The kernels used in the assembly phase for each of the problems is shown in the tables presented in this Appendix.

Kernel	Laplacian	Helmholtz	Advection	Diffusion
`dshape_dot_vector_shape`			✓	
`dshape_tensor_dshape`				✓
`dshape_dot_dshape`	✓	✓		
`shape_shape`		✓	✓	

Table A.1: Kernels implementing bilinear forms and their use in test problems

Kernel	Laplacian	Helmholtz	Advection	Diffusion
`matrix_addto`	✓	✓	✓	✓
`matrix_addto_diffusion`				✓
`matrix_addto_sum_diag`			✓	
`vector_addto`	✓	✓	✓	✓

Table A.2: Kernels implementing addto operations and their use in test problems

Kernel	Laplacian	Helmholtz	Advection	Diffusion
`shape_rhs`	✓	✓		
`advection_rhs`			✓	
`diffusion_rhs`				✓

Table A.3: Kernels which evaluate the RHS of equations and their use in test problems

Kernel	Laplacian	Helmholtz	Advection	Diffusion
`transform_to_physical`	✓	✓	✓	✓
`daxpy`	✓		✓	
`scatter_rhs_values`			✓	✓

Table A.4: Kernels which provide "utility" functions and their use in test problems

Appendix B

UFL Codes for Advection and Diffusion

These codes are written by David Ham. They are included for completeness, and for the reader to refer to in the context of chapter 5.

B.1 Advection

```
T=state.scalar_fields(Tracer)
U=state.vector_fields(Velocity)
UNew=state.vector_fields(NewVelocity)

# We are solving for the Tracer, T.
t=Function(T)
p=TrialFunction(T)
q=TestFunction(T)

#The value of the advecting velocity U is known.
u=Function(U)
unew=Function(UNew)

#Mass matrix.
M=p*q*dx

#Solve for T1-T4.
rhs=dt*dot(grad(q),u)*t*dx
t1=solve(M,rhs)

rhs=dt*dot(grad(q),(0.5*u+0.5*unew))*(t+0.5*t1)*dx
t2=solve(M,rhs)

rhs=dt*dot(grad(q),(0.5*u+0.5*unew))*(t+0.5*t2)*dx
t3=solve(M,rhs)

#Solve for T at the next time step.
rhs=action(M,t) +1.0/6.0*t1 + 1.0/3.0*t2 + 1.0/3.0*t3 + 1.0/6.0*t4
t=solve(M,t)
```

B.2 Diffusion

Assuming the definitions above, the UFL code for the diffusion step is:

```
mu=state.tensor_fields(TracerDiffusivity)
```

```
i,j=indices(2)

M=p*q*dx
d=-grad(q)[i]*mu[i,j]*grad(p)[j]*dx

A=m-0.5*d
rhs=action(M+0.5*d,t)

t=solve(A,rhs)
```

This UFL specification makes use of the Einstein summation convention, in which repeated indices are summed over their range in an expression.

Bibliography

[Advanced Micro Devices, 2008] Inc. Advanced Micro Devices. *ATI Stream SDK User Guide*, 2008.

[Aho *et al.*, 2006] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[Allard *et al.*, 2007] Jérémie Allard, Stéphane Cotin, François Faure, Pierre-Jean Bensoussan, François Poyer, Christian Duriez, Hervé Delingette, and Laurent Grisoni. Sofa an open source framework for medical simulation. In *Medicine Meets Virtual Reality (MMVR'15)*, Long Beach, USA, February 2007.

[Alnaes and Logg, 2009a] Martin Alnaes and Anders Logg. UFL. http://www.fenics.org/wiki/UFL, Retrieved 23 Jul 2009, 2009.

[Alnaes and Logg, 2009b] Martin Alnaes and ANders Logg. UFL Specification and User Manual. http://www.fenics.org/pub/documents/ufl/ufl-user-manual/ufl-user-manual.pdf, Retrieved 15 September 2009, 2009.

[Auer *et al.*, 2006] A.A. Auer, G. Baumgartner, D.E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Krishnamoorthy, S. Krishnan, et al. Automatic code generation for many-body electronic structure methods: the tensor contraction engine. *Molecular Physics*, 104(2):211–228, 2006.

[Bagheri and Scott, 2004] Babak Bagheri and L. Ridgway Scott. About Analysa. Technical Report TR-2004-09, University of Chicago, 2004.

[Balay *et al.*, 2008] Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.0.0, Argonne National Laboratory, 2008.

[Bangerth *et al.*, 2007] W. Bangerth, R. Hartmann, and G. Kanschat. *deal. IIA general-purpose object-oriented finite element library*. ACM New York, NY, USA, 2007.

[Barrett *et al.*, 1994] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[Baumgartner *et al.*, 2002] G. Baumgartner, D.E. Bernholdt, D. Cociorva, R. Harrison, S. Hirata, C.C. Lam, M. Nooijen, R. Pitzer, J. Ramanujam, and P. Sadayappan. A high-level approach to synthesis of high-performance codes for quantum chemistry. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–10. IEEE Computer Society Press Los Alamitos, CA, USA, 2002.

[Beazley, 1996] David M. Beazley. SWIG : An Easy to Use Tool for Integrating Scripting Languages with C and C++ . http://www.swig.org/papers/Tcl96/tcl96.html, Retrieved 9 September 2009, 1996.

[Becker *et al.*, 2009] Aaron Becker, Isaac Dooley, and Laxmikant Kale. Flexible Hardware Mapping for Finite Element Simulations on Hybrid CPU / GPU Clusters. In *SAAHPC : Symposium on Application Accelerators in HPC*, July 2009.

[Bolz *et al.*, 2005] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 171, New York, NY, USA, 2005. ACM.

[Buatois *et al.*, 2007] Luc Buatois, Guillaume Caumon, and Bruno Levy. Concurrent Number Cruncher: An Efficient Sparse Linear Solver on the GPU. In *High Performance Computation Conference (HPCC), Springer Lecture Notes in Computer Sciences*, 2007. Award: Second best student paper.

[Cevahir *et al.*, 2009] Ali Cevahir, Akira Nukada, and Satoshi Matsuoka. Fast Conjugate Gradients with Multiple GPUs. In *9th International Conference on Computational Science*, pages 893–903, 2009.

[Comas *et al.*, 2008] Olivier Comas, Zeike A. Taylor, Jérémie Allard, Sébastien Ourselin, Stéphane Cotin, and Josh Passenger. Efficient Nonlinear FEM for Soft Tissue Modelling and Its GPU Implementation within the Open Source Framework SOFA. In *ISBMS '08: Proceedings of the 4th international symposium on Biomedical Simulation*, pages 28–39, Berlin, Heidelberg, 2008. Springer-Verlag.

[Donea, 2003] J. Donea. *Finite Element Methods for Flow Problems*. Wiley Interscience, 2003.

[Dular and Geuzaine, 2005] P. Dular and C. Geuzaine. *GetDP Reference Manual*, 2005.

[Dupont *et al.*, 2003] T. Dupont, J. Hoffman, C. Johnson, R. C. Kirby, M. G. Larson, A. Logg, and L. R. Scott. The FEniCS project. Technical Report 2003–21, Chalmers Finite Element Center Preprint Series, 2003.

[Farrell *et al.*, 2009] Patrick Farrell, Colin Cotter, and Matthew Piggott. Fluidity Test Cases. http://amcg.ese.ic.ac.uk/index.php?title=Test_cases, Retrieved 11 September 2009, 2009.

[Filipovic *et al.*, 2009a] 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.

[Filipovic *et al.*, 2009b] 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 - Poster Session*, July 2009.

[Gorman *et al.*, 2008] Gerard Gorman, Matthew Piggot, and Patrick Farrell. About Fluidity. http://amcg.ese.ic.ac.uk/index.php?title=FLUIDITY, 2008.

[Gschwind *et al.*, 2006] Michael Gschwind, H. Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, 26(2):10–24, 2006.

[Ham *et al.*, 2009] David Ham, Matthew Piggot, and Christopher Pain. Fluidity/ICOM Manual, revision 11245. http://amcg.ese.ic.ac.uk/cgi-bin/viewvc.cgi/trunk/manual/fluidity_manual.pdf?revision=11245&root=fluidity&pathrev=11245, Retrieved 10 September 2009, 2009.

[Ham, 2009a] David Ham. The Femtools Manual, Revision 10934. http://amcg.ese.ic.ac.uk/cgi-bin/viewvc.cgi/trunk/femtools/doc/femtools_manual.pdf?revision=10934&root=fluidity, Retrieved 11 September 2009, 2009.

[Ham, 2009b] David A. Ham. Documentation for the Advection-Diffusion Test Problem. http://www.doc.ic.ac.uk/~grm08/testad.pdf, Retrieved 14 September 2009, 2009.

[Hecht *et al.*, 2005] F. Hecht, O. Pironneau, A. L. Hyaric, and K. Ohtsuka. *FreeFEM++ Manual*, 2005.

[Howes *et al.*, 2009a] Lee Howes, Anton Lokhmotov, Alastair F. Donaldson, and Paul H.J. Kelly. Decoupled Access/Execute metaprogramming for GPU-accelerated systems. In *Symposium on Application Accelerators in High Performance Computing (SAAHPC)*, 2009.

[Howes *et al.*, 2009b] 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.

[Khronos Group, 2008] The Khronos Group. *OpenCL 1.0 Working Specification*, 2008.

[Klöckner *et al.*, 2009] A. Klöckner, T. Warburton, J. Bridge, and J.S. Hesthaven. Nodal discontinuous Galerkin methods on graphics processors. *Journal of Computational Physics*, In Press:–, 2009.

[Klöckner, 2009] Andreas Klöckner. PyCUDA Documentation. http://mathema.tician.de/software/pycuda, Retrieved 9 September 2009, 2009.

[Komatitsch *et al.*, 2009] 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.

[Langtangen, 2003] Hans Petter Langtangen. *Computational Partial Differential Equations, Numerical Methods and Diffpack Programming*. Springer-Verlag, 2003.

[Lindholm *et al.*, 2008] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, 2008.

[Logg and Alnaes, 2009] Anders Logg and Martin Alnaes. The FEniCS Project. http://www.fenics.org, Retrieved 9 September 2009, 2009.

[Logg and Wells, 2009] A. Logg and G. N. Wells. DOLFIN: Automated finite element computing. http://www.fenics.org/pub/documents/dolfin/papers/dolfin-2009.pdf, Retrieved 15 September 2009, 2009.

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

[Long, 2003] K.R. Long. Sundance rapid prototyping tool for parallel PDE optimization. *Large-scale PDE-constrained optimization*, page 331, 2003.

[Markall and Kelly, 2009] 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.

[Markall *et al.*, 2009] Graham Markall, David A. Ham, and Paul H. J. Kelly. Fitting the Ocean on to a Graphics Card: Towards Running ICOM on Massively Parallel Processors. Presented at the 8th International Workshop on Unstructured Mesh Numerical Modelling of Coastal, Shelf and Ocean Flows, September 2009.

[Markall, 2009] Graham Markall. Porting Fortran to CUDA. https://spo.doc.ic.ac.uk/twiki/bin/view.cgi/External/PortingFortranToCUDA, Retrived 9 September 2009, 2009.

[Miller *et al.*, 2007] Karol Miller, Grand Joldes, Dane Lance, and Adam Wittek. Total Lagrangian explicit dynamics finite element algorithm for computing soft tissue deformation. *Communications in Numerical Methods and Engineering*, 23(2):121–134, 2007.

[Monk, 2003] Peter Monk. *Finite Element Methods for Maxwell's Equations*. Clarendon, 2003.

[NVidia, 2007] NVidia. *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[NVidia, 2009a] NVidia. NVIDIA CUDA Reference Manual, Version 2.2. http://developer.download.nvidia.com/compute/cuda/2_2/toolkit/docs/CUDA_Reference_Manual_2.2.pdf, Retrieved 10 September 2009, 2009.

[NVidia, 2009b] NVidia. The Cg Homepage. http://developer.nvidia.com/page/cg_main.html, Retrieved 9 September 2009, 2009.

[NVidia, 2009c] NVidia. The CUDA Occupancy Calculator. http://news.developer.nvidia.com/2007/03/cuda_occupancy_.html, Retrieved 9 September 2009, 2009.

[Ogden, 1997] R. W. Ogden. *Non-Linear Elastic Deformations*. Dover Publications, 1997.

[Perryman and Kelly, 2008] Tristan Perryman and Paul H. J. Kelly. Accelerating Fluidity Using the GPU. UROP Report, Imperial College London, 2008.

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

[Quinlan *et al.*, 2009b] Dan Quinlan, Chunhua Liao, Thomas Panas, Jeremiah Willcock, Robb Matzke, Markus Schordan, Rich Vuduc, and Qing Yi. ROSE HTML Reference. http://www.rosecompiler.org/ROSE_HTML_Reference/index.html, Retrieved 15 September 2009, 2009.

[Reid, 2009] John K. Reid. Interoperability [of Fortran] with C. http://www.fortran.bcs.org/2002/interop.htm, Retrieved 9 September 2009, 2009.

[Rumpf and Strzodka, 2001] M. Rumpf and R. Strzodka. Nonlinear diffusion in graphics hardware. In *In Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym 01*, pages 75–84. Springer, 2001.

[Seiler *et al.*, 2008] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, August 2008.

[Sherwin *et al.*, 2009] Spencer Sherwin, Ian Matthews, and Colin Cotter. *Finite Element Methods, MSc. Short Course*. Department of Aeronautics, Imperial College London, 2009.

[Shewchuk, 1994] Jonathan R Shewchuk. An Introduction to the Conjugate Gradient Method Without the Agonizing Pain. Technical report, Pittsburgh, PA, USA, 1994.

[Silva, 2005] Malik Silva. Sparse matrix storage revisited. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 230–235, New York, NY, USA, 2005. ACM.

[Taylor *et al.*, 2007] Zeike A. Taylor, Mario Cheng, and Sébastien Ourselin. Realtime Nonlinear Finite Element Analysis for Surgical Simulation Using Graphics Processing Units. In *In Proceedings of the 10th International Conference on Medical Image Computing and Computer Assisted Intervention (LNCS 4791)*, pages 701–708. Springer Berlin/Heidelberg, 2007.

[Taylor *et al.*, 2008] Z.A. Taylor, M. Cheng, and S. Ourselin. High-Speed Nonlinear Finite Element Analysis for Surgical Simulation Using Graphics Processing Units. *Medical Imaging, IEEE Transactions on*, 27(5):650–663, May 2008.

[Volkov and Demmel, 2008] Vasily Volkov and James W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[Wang *et al.*, 2009] Mingliang Wang, Hector Klie, Manish Parashar, and Hari Sudan. Solving Sparse Linear Systems on NVIDIA Tesla GPUs. In *9th International Conference on Computational Science*, volume 5544 of *Lecture Notes in Computer Science*, pages 864–873. Springer, 2009.

[Weisstein, 2009] Eric W. Weisstein. The Runge-Kutta Method. http://mathworld.wolfram.com/Runge-KuttaMethod.html, Retrieved 9 September 2009, 2009.

[Wiggers *et al.*, 2007] W. A. Wiggers, V. Bakker, A. B. J. Kokkeler, and G. J. M. Smit. Implementing the conjugate gradient algorithm on multi-core systems. In J. Nurmi, J. Takala, and O. Vainio, editors, *Proceedings of the International Symposium on System-on-Chip (SoC 2007), Tampere*, pages 11–14, Piscataway, NJ, November 2007. IEEE.