Imperial College London
Department of Computing

Multilayered Abstractions for Partial Differential Equations

Graham Robert Markall

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing and the Diploma of Imperial College, June 2013

Abstract

How do we build maintainable, robust, and performance-portable scientific applications? This thesis argues that the answer to this software engineering question in the context of the finite element method is through the use of layers of Domain-Specific Languages (DSLs) to separate the various concerns in the engineering of such codes.

Performance-portable software achieves high performance on multiple diverse hardware platforms without source code changes. We demonstrate that finite element solvers written in a low-level language are not performance-portable, and therefore code must be specialised to the target architecture by a code generation framework. A prototype compiler for finite element variational forms that generates CUDA code is presented, and is used to explore how good performance on many-core platforms in automatically-generated finite element applications can be achieved. The differing code generation requirements for multi- and many-core platforms motivates the design of an additional abstraction, called PyOP2, that enables unstructured mesh applications to be performance-portable.

We present a runtime code generation framework comprised of the Unified Form Language (UFL), the FEniCS Form Compiler, and PyOP2. This toolchain separates the succinct expression of a numerical method from the selection and generation of efficient code for local assembly. This is further decoupled from the selection of data formats and algorithms for efficient parallel implementation on a specific target architecture.

We establish the successful separation of these concerns by demonstrating the performance-portability of code generated from a single high-level source code written in UFL across sequential C, CUDA, MPI and OpenMP targets. The performance of the generated code exceeds the performance of comparable alternative toolchains on multi-core architectures.

Acknowledgements

I would like to express my thanks and appreciation to the following people, whose encouragement and support made it possible for me to complete this thesis:

- Paul Kelly, My supervisor, who has guided and shaped my work since I first arrived at Imperial College. His enthusiasm, patience, and dedication have been unwavering.

- David Ham, My second supervisor, who has dedicated an extraordinary amount of time and effort to share the benefit of his knowledge and experience with me throughout my studies.

- Sarah, my patient, loving, and long-suffering wife. Her constant support and encouragement have kept me going through my low points, and have made the high points immeasurably more enjoyable.

- Florian Rathgeber, whose meticulous attention to detail has inspired me to care for and craft the software that I write, and for helping me to judge my work in a fair light.

- Francis Russell, who has kindly shared much of his understanding of the finite element method and the PhD process with me, as well as taking my mind off things by participating in many enjoyable discussions.

- Gerard Gorman and Peter Jimack, my examiners, for interesting discussions about this thesis and their suggestions for its improvement.

- Carlo Bertolli, with whom I have had many interesting discussions about this work and other topics, and whose sense of humour I have enjoyed very much.

- Stuart Archibald and James Kimber, who have both spent a great deal of time listening to me waffle on about my research, and also for encouraging me to be involved in the Imperial College Linux Users' Society.

Contents

Chapter 1

Introduction

1.1 Thesis Statement

This thesis argues that multiple layers of domain-specific abstractions are an essential requirement for the development of maintainable, robust, and performance-portable finite element codes. For software to be performance-portable, it must be able to be run on multiple different architectures with good performance on all of them without the need for manual platform-specific performance optimisations.

1.2 Motivation and Objectives

The FEniCS project [LMW+12] enables the development of maintainable and robust finite element codes. It has shown that the *Unified Form Language* (UFL) provides an appropriate abstraction of the finite element method for generating efficient code from maintainable sources, and allows the expression of a wide variety of variational forms. UFL allows the user to declaratively specify a finite element discretisation instead of writing the code that implements it in an imperative style. As we will show, using UFL for writing finite element codes is desirable as it prevents common errors and eliminates many time-consuming and error-prone tasks required when developing in a low-level language.

The prevalence and availability of diverse multi- and many-core architectures prompts investigation to determine the optimal implementation strategies on these devices. Although significant speedups have been demonstrated for finite element applications [FPF09a, KME09, KGEM10, CLD11], it is necessary to manually tune codes in order to obtain optimal performance on each platform. Transformations such as loop unrolling, loop tiling, common subexpression elimination, and software pipelining, amongst many other well-documented techniques [ALSU06], are used. Compilers can perform some of these automatically, but it is necessary for scientists to devote a considerable amount of time and effort to hand-optimise code.

Although these optimisations often result in significant performance gains, the differences between the designs of different architectures dictate that the required set of optimisations varies. Thus, there are no performance-portable finite element frameworks. We therefore seek to facilitate the development of maintainable, robust, and performance portable finite element codes. This thesis lays out the motivation for, and charts the development of a framework that meets this objective.

1.3 Application

The work presented in this thesis forms the basis of an effort to improve the long-term sustainability of the Fluidity computational fluid dynamics code [App10] through the application of code generation technology. This will support the enhancement of Fluidity in two ways:

- In the short-term, this work enables user extensibility of the numerical methods used by Fluidity. Users can specify new discretisations in UFL as part of the simulation parameters. The user-specified methods are incorporated into their simulation at runtime.

- The long-term goal is to re-engineer the core of Fluidity using the abstractions developed in this thesis. This work will be influenced by the experience gained from the incorporation of user-provided methods. This goal is not addressed within the scope of this thesis.

1.4 Contributions

- In Chapter 4 we demonstrate that a single source written in a low-level language cannot be performance-portable. This is shown by exploring the changes in data format and global assembly algorithm that are required to get the best performance out of different architectures. We demonstrate that the optimal choice varies both with the target architecture and with the problem parameters.

- In Chapter 5, we present the design and implementation of a form compiler that generates code for many-core platforms. We also present algorithms that minimise the code size of local assembly kernels. This development exemplifies the requirements for a toolchain that supports efficient code generation for many-core platforms, and provides guidance for the design of a performance-portable parallel execution layer for a finite element form compilation toolchain.

- In Chapter 6 we extend the OP2 abstraction to support linear algebra and runtime code generation in order to use it as a performance-portable parallel execution layer for finite element computations. The design of this framework, called PyOP2, is guided by the conclusions of Chapters 4 and 5.

- In Chapter 7 we use PyOP2 as part of a framework for building maintainable, robust, and performance-portable finite element solvers. The performance portability of the framework is demonstrated by showing that performance exceeding the best available alternative is obtained on different multi- and many-core platforms.

1.5 Statement of Originality

I hereby declare that this document is the result of my own work as is the work it presents except where otherwise stated.

1.6 Dissemination

The work contained within this thesis has been disseminated to a wider community through publications, presentations, and the release of software under open-source licences. The list of publications is as follows:

[MSH+13] **Finite element assembly strategies on multi-core and many-core architectures.** This paper is a journal version of [MHK10]. It presents the exploration of the finite element assembly implementation space described in Chapter 4.

[GMS+11] **Performance Analysis and Optimization of the OP2 Framework on Many-Core Architectures.** This paper is the journal version of [GMS+10]. The work on OP2 provides a basis for the PyOP2 implementation described in Chapter 6.

[MRM+13] **Performance-portable finite element assembly with PyOP2 and FEniCS.** This paper is a conference version of [RMM+12]. It presents an overview of the framework and the demonstration of its performance portability that is presented here in Chapter 7.

The toolchains described in this thesis were presented to the FEniCS conferences on the following occasions:

FEniCS '12. The Manycore Form Compiler described in Chapter 5 was presented at the FEniCS '12 conference.

FEniCS '13. PyOP2 and the Firedrake framework described in Chapters 6 and 7 was presented at the FEniCS '13 conference.

The software tools developed in this thesis are hosted at the following locations:

The Manycore Form Compiler. Available at `https://github.com/gmarkall/manycore_form_compiler`.

FFC-PyOP2. Available at https://bitbucket.org/mapdes/ffc. This work will be proposed for merge into the main FFC repository in the near future.

PyOP2. Available at https://github.com/OP2/PyOP2.

1.7 Thesis outline

In Chapters 2 and 3 a review of relevant literature is formed. The material covered in Chapter 2 is referenced in later chapters, and forms a basis for the work in this thesis. Chapter 3 surveys related work that is relevant for understanding how scientific applications can be optimised on many-core architectures, and for understanding how performance-portability is achieved in other domains.

In Chapter 4, we explore how changes in data format and global assembly algorithm affect the performance of finite element applications on multi- and many-core architectures. This exploration demonstrates that the optimal choice of these parameters varies with the target hardware, and therefore a single implementation cannot be performance-portable.

Chapter 5 describes the Manycore Form Compiler, a prototype finite element form compiler. This compiler was used to discover the requirements for a parallel execution layer in a form compilation toolchain, and a number of recommendations are presented at the end of the chapter.

These recommendations are incorporated into the design of PyOP2, which is described in Chapter 6. PyOP2 provides a framework for building parallel unstructured mesh applications in Python, whilst abstracting the target hardware beneath its API.

In Chapter 7 we describe how PyOP2 is integrated into a finite element code generation toolchain. The generated code is evaluated on multi- and many-core platforms to demonstrate the performance-portability of the toolchain.

Chapter 2

Background

2.1 Introduction

This chapter begins by familiarising the reader with relevant aspects of the finite element method, with a focus on the algorithms and data structures used in the implementation of the method. The mathematical theory of the method is covered in brief for the purpose of establishing our notation and terminology. This leads into a discussion of the FEniCS toolchain and its components, which provide a flexible environment for the development of finite element solvers using a DSL.

Subsequently, we discuss the differences between modern parallel architectures, which include multi-core CPUs and many-core GPUs, in order to convey the challenge they pose to writing performance portable software, and to examine how good performance is achieved on them. We introduce the Æcute framework, which is aimed at providing high performance on these architectures by decoupling the specification of a computation and the data it operates on. This leads in to a discussion of the OP2 framework which can be seen as a generalisation of Æcute to parallel unstructured mesh applications.

2.2 The Finite Element Method

Here we provide a brief overview of the mathematical formulation of the finite element method. For a complete treatment, see [KS99, CMP01]. The strong form of a partial differential equation is:

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

where L is any differential operator, u is a dependent variable, and q is a known function that does not depend on u. For simplicity we will present the case in which L is linear, however everything here, and in this thesis, is applicable to the nonlinear case. Take for example, $L \equiv -\nabla^2$ and q is the source term in Poisson's equation. Solving this equation numerically gives a solution u^δ, which may not satisfy Equation 2.1 exactly. So we have:

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

where R is the *residual*. This provides a measure of the amount by which the numerical solution fails to satisfy Equation 2.1. In the ideal case, $R = 0$, and the numerical solution is the exact solution, so we must try to eliminate this term. However, requiring the numerical solution to be exact makes it too difficult to find a solution to most problems. If we are prepared to tolerate some inaccuracy, we can transform the system to weaken the definition of equality. First, we weight the equation with a *test function*, v, and then integrate over the whole domain, Ω, giving:

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

Now, we make the assumption that $R \simeq 0$, denoting that u^δ solves this system in an "average"

sense, since the two integrals on the right-hand side are equal. This leaves:

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

This form is known as the *integral form* of Equation 2.1. We can then alter this system to lower the order of derivatives under the integral sign and to introduce natural boundary conditions, by applying integration by parts. This results in:

$$\int_{\partial\Omega} vl(u^\delta)\cdot\mathbf{n}\,\mathrm{d}s - \int_\Omega \nabla v\cdot l(u^\delta)\,\mathrm{d}X = \int_\Omega vq\,\mathrm{d}X, \tag{2.5}$$

where $l(u^\delta)$ is defined such that $\nabla l(u^\delta) = L(u^\delta)$, and $\mathbf{n}$ is the outward-pointing normal to the domain. Equation 2.5 is the *weak form* of the differential equation. The reduced order of the derivative in the weak form lowers the order of continuity required for the derivatives of the basis of the numerical solution u^δ.

We choose the space for the approximation u^δ as a finite-dimensional subspace of the space of u. This space is known as the *trial space*. The approximation takes the form:

$$u^\delta = \sum_{i=1}^{N_{dof}} \hat{u}_i\Phi_i, \tag{2.6}$$

where N_{dof} is the number of basis functions in the space. The functions Φ_i are known as the *trial functions*. We also choose the test space as a finite dimensional subspace of the space of u:

$$v = \sum_{j=1}^{N_{dof}} \hat{v}_j v_j. \tag{2.7}$$

In the Galerkin method, test functions are chosen such that $v_j = \Phi_j$, i.e. the test and trial spaces are the same. Other choices of test space may be made, and some examples are given in

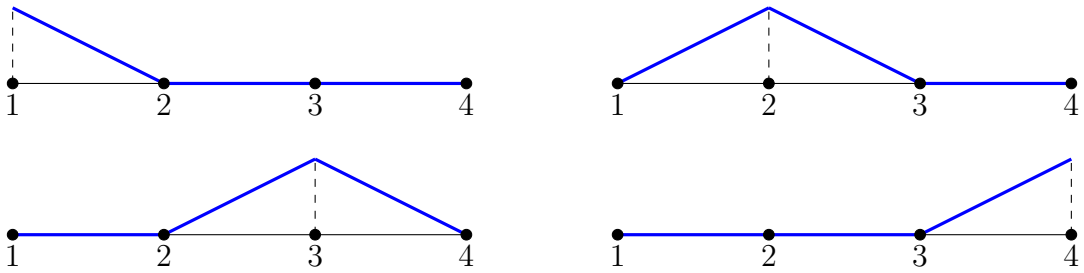


Figure 2.1: A piecewise continuous linear basis over a one-dimensional domain with four nodes and three elements.

[KS99] and [CMP01]. The Finite Element Method uses basis functions that are local, i.e. have compact support. Throughout this thesis, Galerkin Finite Element Methods are considered.

2.2.1 An Example Basis

An example of a choice of basis functions Φ_i for the Galerkin Finite Element Method for a one-dimensional domain is a piecewise continuous linear basis:

$$\Phi_i(x_i) = 1, \text{ and } \forall k : i \neq k, \Phi_i(x_k) = 0, \tag{2.8}$$

where x_i is the i-th node in a set of nodes placed at discrete points in the domain. We can define the basis functions as follows:

$$\Phi_i = \begin{cases} \frac{x - x_{k-1}}{x_k - x_{k-1}} & \text{if } x \in [x_{k-1}, x_k] \\ \frac{x_{k+1} - x}{x_{k+1} - x_k} & \text{if } x \in [x_k, x_{k+1}] \\ 0 & \text{otherwise.} \end{cases} \tag{2.9}$$

An example of this basis for a four-node domain is shown in Figure 2.1.

2.2.2 Assembly

The weak form may be solved for u^δ by performing an assembly procedure that transforms it into a system of linear equations. This procedure consists of two stages:

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

Global Assembly. The local matrices, $\mathbf{M}^e$, and vectors, $\mathbf{b}^e$, are used to form a *global matrix*, $\mathbf{M}$, and *global vector*, $\mathbf{b}$, respectively. This process couples the contributions of elements together. The sparsity structure of the global matrix depends on the connectivity of the mesh, and as such it is typically very sparse. The Compressed Sparse Row (CSR) [BBC+94] format is often used to reduce the storage requirement of the matrix and to eliminate redundant computations. This process is discussed in more detail in Chapter 4.

The global matrix and vector are used to form a linear system,

$$A\mathbf{x} = \mathbf{b}, \tag{2.10}$$

where the solution x is the coefficients of the basis (i.e. $\hat{u}_i$) representing the solution to the discretised problem. The system is often solved using an iterative method, such as the Conjugate Gradient or GMRES methods [BBC+94], due to the size and sparsity of the system. The use of an iterative method requires repeated computation of the *sparse matrix-vector product* (SpMV) $\mathbf{y} = \mathbf{M}\mathbf{v}$.

2.2.3 Variation in Finite Element Methods

Two key choices can be made that result in different finite element discretisations of the original equation:

1. The weak form of the equation.
2. The type of basis functions.

Making these choices results in discretisations with different properties. We do not cover the mathematical theory behind the discretisations or basis functions in this thesis, but we stress that it is important for scientists to be able to easily vary these choices in order to test and develop finite element solvers. We provide some examples of different choices of weak form and discretisation in Section 2.2.5.

2.2.4 Boundary Conditions

Appropriate boundary conditions must be enforced in order to produce a unique solution to a differential equation. Boundary conditions are usually one of two kinds:

Dirichlet. These specify the exact value of the solution, or components of it, at a boundary node. Examples of Dirichlet boundary conditions include the no-slip condition, $\mathbf{u} = 0$, which states that the velocity of a fluid is zero at a boundary, and the free-slip condition, $\mathbf{u} \cdot \mathbf{n} = 0$ where $\mathbf{n}$ is the unit normal, which states that fluid moves freely along a boundary, but not through it. The implementation of Dirichlet conditions involves modifying particular matrix rows such that their off-diagonal entries are equal to zero, and their diagonal entries are equal to one. The modification of particular values in the right-hand side vector are also required.

Neumann. These specify the value of the derivative of a function at a boundary node. An example of a Neumann boundary condition is the "do-nothing" boundary condition, $\frac{\partial \mathbf{u}}{\partial X} = 0$,

which prescribes free flow out of the domain. The implementation of Neumann boundary conditions in the finite element method requires an assembly process over the edges of the domain, as it involves the boundary integral in the weak form (Equation 2.5). As a result, this computation uses a domain of dimension $n-1$ when the domain is of dimension n. For example, in a 2D domain, 1D line integrals are evaluated over the boundaries, and in a 3D domain, 2D surface integrals are evaluated over the boundaries.

2.2.5 Examples

The finite element method is often coupled with other numerical methods. For problems with a time-varying solution, it is usual to couple the finite element method with another scheme for discretising the time derivative. Equations with nonlinear terms require coupling with a nonlinear solver such as a Newton solver [Kel95]. When solving the Navier-Stokes equations with the finite element method, the projection method is often used in order to decouple the solution of the pressure and velocity fields [CM90].

The examples we present below illustrate some of the choices that can be made when solving an equation using the finite element method. These examples are reproduced from the FEniCS demos, which are located in `/usr/share/dolfin/demo` on any system on which the FEniCS toolchain is installed (we discuss the FEniCS Project in Section 2.4).

Poisson's Equation

Poisson's Equation is:

$$-\nabla \cdot \nabla u = f, \tag{2.11}$$

where $u = u(x, y)$ is function of space, and $f = f(x, y)$ is a forcing function. The weak form, after integration by parts to lower the order of the derivative in the term involving $\nabla \cdot \nabla u$ is:

$$\int_\Omega \nabla v \cdot \nabla u \, \mathrm{d}X = \int_\Omega v f \, \mathrm{d}X + \int_{\partial\Omega} v \nabla u \cdot \mathbf{n} \, \mathrm{d}s, \tag{2.12}$$

where we have introduced the test function v and u is the trial function. An appropriate basis for the solution is a Lagrange basis with polynomial order $p = 1$. Since there is no time derivative or nonlinear term in the equation, this form of the equation may be solved by performing the assembly procedure and solving the resulting system of equations.

Next, we consider a mixed formulation of Poisson's equation. This has the form:

$$\sigma - \nabla u = 0, \tag{2.13}$$

$$\nabla \cdot \sigma = f. \tag{2.14}$$

The weak form of these equations is:

$$\int_\Omega \sigma \cdot \tau + u \nabla \cdot \tau + v \nabla \cdot \sigma \, \mathrm{d}X = -\int_\Omega v f \, \mathrm{d}X, \tag{2.15}$$

where we define test functions τ and v, and u and σ are trial functions. Brezzi-Douglas-Marini elements of degree k are used for the space which σ and τ occupy, and discontinuous Galerkin elements of order $k - 1$ are used for the space for v and u [KLRT12].

We note how modifications to the equation being solved result in changes to the weak form, and the choice of basis functions. This demonstrates how flexibility in the weak form and basis functions is necessary for solving a wide range of problems.

The Advection-Diffusion Equation

A general form of the Advection-Diffusion equation is:

$$\frac{\partial T}{\partial t} = \nabla \cdot (D \nabla T) - \nabla \cdot (\mathbf{u} T) + R, \tag{2.16}$$

where T is the concentration of some tracer in a fluid with velocity $\mathbf{u}$ and diffusivity D that evolves over time t and has a source R. If we assume that the velocity field is divergence-free, the source term is zero, and diffusivity is both constant and isotropic, Equation 2.16 simplifies to:

$$\frac{\partial T}{\partial t} = \underbrace{D \nabla^2 T}_{\text{Diffusion}} - \underbrace{\mathbf{u} \cdot \nabla T}_{\text{Advection}}, \tag{2.17}$$

where we refer to the marked terms as the advection term and the diffusion term. The weak form of Equation 2.17 after integration by parts of the advection and diffusion terms is:

$$\int_\Omega q \frac{\partial T}{\partial t} \, \mathrm{d}X = \int_{\partial\Omega} q (D \nabla T - \mathbf{u} T) \cdot \mathbf{n} \, \mathrm{d}s - \int_\Omega D \nabla q \cdot \nabla T \, \mathrm{d}X + \int_\Omega \nabla q \cdot \mathbf{u} T \, \mathrm{d}X. \tag{2.18}$$

Only the spatial derivatives are discretised using the finite element method in this example. Although it is possible to discretise time with the finite element method, it is presently uncommon to do so in practice. Discretising the time derivative with a theta scheme [HNW93] where $\theta = \frac{1}{2}$ (which is also known as the Crank-Nicolson scheme) yields the following:

$$\begin{aligned} &\int_\Omega q T^{n+1} \, \mathrm{d}X - \frac{\Delta t}{2} \left(\int_{\partial\Omega} q (D \nabla T^{n+1} - \mathbf{u} T^{n+1}) \cdot \mathbf{n} \, \mathrm{d}s - \int_\Omega D \nabla q \cdot \nabla T^{n+1} \, \mathrm{d}X + \int_\Omega \nabla q \cdot \mathbf{u} T^{n+1} \, \mathrm{d}X \right) \\ &= \int_\Omega q T^n \, \mathrm{d}X + \frac{\Delta t}{2} \left(\int_{\partial\Omega} q (D \nabla T^n - \mathbf{u} T^n) \cdot \mathbf{n} \, \mathrm{d}s - \int_\Omega D \nabla q \cdot \nabla T^n \, \mathrm{d}X + \int_\Omega \nabla q \cdot \mathbf{u} T^n \, \mathrm{d}X \right). \end{aligned} \tag{2.19}$$

```
E = FiniteElement("Lagrange", "triangle", 1)

v = TestFunction(E)
u = TrialFunction(E)
f = Coefficient(E)
n = f.cell().n

poisson = dot(grad(v),grad(u))*dx == v*f*dx + v*dot(grad(u),n)*ds
```

Figure 2.2: UFL code for the weak form of Poisson's equation

.

This form of the equation can be advanced in time by performing the finite element assembly process and solving the resulting system. The solution is advanced in time by an arbitrary amount by repeatedly performing this process, each time using the solution from the previous timestep in the assembly at the present time. As there is no previous solution available at the first timestep, an initial condition must be defined that is used as its input.

2.2.6 Remarks

In the examples that have been presented, we have seen that the variation of finite element methods is achieved through the choice of weak form and basis functions. In practice, the finite element method is often coupled with other numerical methods and techniques. Although we have presented only a small number of examples, it is important to note that active research efforts involve developing new basis functions, weak forms, and couplings with other numerical methods.

2.3 The Unified Form Language

The *Unified Form Language* (UFL) is part of the FEniCS toolchain, which we will discuss in Section 2.4. It provides a means for the specification of variational forms and the basis functions used to discretise them. This allows for the easy translation of the mathematical representation of finite element methods into computer code. To exemplify the language, we present the translation of the example forms in the previous section into UFL.

```
g = Coefficient(E)
L = v*f*dx + v*g*ds
```

Figure 2.3: UFL code for specifying a Neumann boundary when solving Poisson's equation.

The translation of Equation 2.12 into UFL is shown in Figure 2.2. The first line of the example declares the type of elements that are to be used. In this case, triangular cells with order 1 Lagrange functions are used. Many other commonly used basis functions are available, and are documented in [KLRT12]. The name given to specify the basis functions has no semantics in UFL itself, but is passed on to the compiler of UFL forms in order to allow it to invoke a finite element tabulator for generating function values for the given basis. The following four lines declare the variables that are to be used in the forms. The UFL keyword `Coefficient` declares the coefficients of basis functions for a field. `n` is the outward-pointing normal to the domain.

The weak form is expressed as an equation in the final line. Note the similarity between the equation as written, and in the code. This similarity makes UFL a powerful tool for rapidly constructing finite element solvers.

In practice if a Neumann boundary condition were to be applied, the term `dot(grad(u),n)` would be replaced by a coefficient specifying the value of the derivative at the boundary. The form `L` would then be specified as shown in Figure 2.3. The coefficient g holds the value of the derivative of the field at the boundary, which is equivalent to $\nabla u \cdot \mathbf{n}$.

The translation of the advection-diffusion form from Equation 2.19 is shown in Figure 2.4. To simplify the example, a Neumann boundary condition $\nabla u = 0$ is applied, which causes the surface integrals to vanish. This models the case where no tracer enters or exits the domain through the boundary. Several terms in the left- and right-hand sides of the equation are identical with the exception that the left-hand side is defined with T^{n+1} and the right-hand side is defined with T^n. The UFL `action` function saves these definitions from being repeated. The declaration of f is written in terms of the trial function `p` (in the code) or T^{n+1} (in the equation). This is used straightforwardly in defining the left-hand side, `a`. For the right-hand side, the *action* of the form `M-0.5*f` on the function `t` is computed. This replaces all instances of the trial function in the form with `t` (T^n), producing the right-hand side of Equation 2.19.

```
T = FiniteElement("Lagrange", "triangle", 1)
U = VectorElement("Lagrange", "triangle", 1)

q = TestFunction(T)
p = TrialFunction(T)
t = Coefficient(T)
v = Coefficient(V)

dt = 0.1 # timestep, delta t
d  = 0.1 # diffusivity

M = p*q*dx
f = dt*(d*dot(grad(q),grad(p))-dot(grad(q),v)*p)*dx

a = M+0.5*f
L = action(M-0.5*f, t)

adv_diff = a == L
```

Figure 2.4: UFL code implementing the weak form of the advection-diffusion equation with Crank-Nicolson timestepping. A Neumann boundary condition $\nabla u = 0$ is applied so that the surface integrals are equal to zero.

This UFL code alone does not implement a complete solver for the equation. An imperative Python implementation is required for the execution of the timestepping scheme. UFL code can be embedded in Python code, so it is unnecessary to split the UFL and Python portions into separate source files.

2.3.1 Remarks

UFL provides a powerful tool for capturing the domain of variability in finite element methods. Mathematical objects expressed in UFL are free of extraneous details, which simplifies the process of translating an equation into a UFL expression.

UFL does not by itself provide a complete environment for the development of finite element solvers, since there are numerous other components to any solver, such as compilation of the variational forms to low-level code, managing the allocation and movement of data, timestepping, and other book-keeping operations. The UFL library is implemented as an embedded Domain-Specific Language (eDSL) in Python that can be integrated into a finite element problem-solving environment.

2.4 The FEniCS Project

FEniCS [LMW+12] is a toolchain that is widely used by scientists and engineers for developing finite element models. Figure 2.5 gives an overview of the interactions between the FEniCS components. DOLFIN provides a complete environment for developing a finite element solver, which can be used either from its C++ interface or from its Python interface. From the programmer's point of view, it can be seen as a library that provides support for the mesh data structures, file I/O, finite element assembly, and a transparent interface to code-generation tools and linear algebra libraries.

The implementation of DOLFIN abstracts many of the details of these components away, which gives the ability to write a finite element solver using a very small amount of code with very little overhead on top of the specification of variational forms. Figure 2.6 gives an example of the complete source code required to write a solver for Poisson's equation described above using DOLFIN. The UFL code that was presented earlier is embedded within the DOLFIN-based solver.

When a form is to be assembled, DOLFIN calls a form compiler that performs just-in-time compilation of the forms and loads them back into the Python interpreter. The form compiler generates C++ code conforming to the *Unified Form Assembly Code* (UFC) specification, rather than directly generating machine code. The Instant library provides facilities for dynamically compiling C++ code into a Python extension and linking it back into the running interpreter. It also provides caching so that identical code is not recompiled on subsequent runs. As well as being called by the form compiler, Instant is also used by DOLFIN for just-in-time compilation of expressions other than variational forms.

The Form Compiler makes use of the Finite Element Automatic Tabulator (FIAT), which tabulates the values of the basis functions at the quadrature points for the elements used in a form. These values are incorporated into the C++ code generated by the compiler for the local assembly functions. In order to add support for new types of element into the toolchain, one has to extend FIAT to generate the values for the new basis. These values can then be

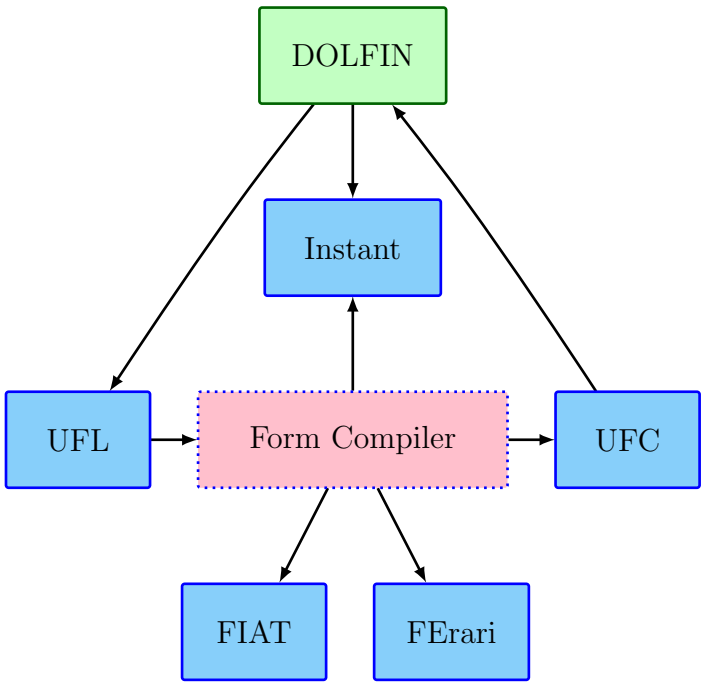


Figure 2.5: Architecture of the FEniCS/DOLFIN problem-solving environment. The Form Compiler can be interchanged to generate different variants of local assembly code.

incorporated into the rest of the toolchain without further modifications being made.

2.4.1 Parallel Execution

DOLFIN presently supports distributed-memory parallelism using MPI and shared memory parallelism using OpenMP. Benchmarks have shown scaling close to linear for up to 1024 processors for an implementation of a CFD solver developed using DOLFIN [JHJ12].

There is presently no support for performing finite element assembly on many-core architectures. However, the performance of local assembly kernels that implement FEniCS's tensor-based assembly strategy [KL12] on GPUs has been investigated [KT13]. This is discussed further in Chapter 3.

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquare(32, 32)
V = FunctionSpace(mesh, "Lagrange", 1)

# Define Dirichlet boundary (x = 0 or x = 1)
def boundary(x):
    return x[0] < DOLFIN_EPS or x[0] > 1.0 - DOLFIN_EPS

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

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Expression("10*exp(-(pow(x[0] - 0.5, 2) + pow(x[1] - 0.5, 2)) / 0.02)")
g = Expression("sin(5*x[0])")
a = inner(grad(u), grad(v))*dx
L = f*v*dx + g*v*ds

# Compute solution
u = Function(V)
solve(a == L, u, bc)
```

Figure 2.6: Finite element solver for Poisson's Equation written using DOLFIN. Note that this is the complete user-written code.

2.5 Fluidity

Fluidity [App10] is a computational fluid dynamics package that predominantly uses the finite element method to solve the field equations in ocean simulations. It has a mature distributed-memory parallel implementation that uses MPI and has been shown to scale up to tens of thousands of cores. More recently, OpenMP has been used to implement shared-memory parallelism in Fluidity [Mit13].

The majority of Fluidity is written in Fortran 90, but access to some of the mesh and field data structures is also provided through a Python interface. The arrays containing the field data are wrapped in Numpy [Oli06] arrays in order to allow the Python code to modify the field values in-place. This interface allows additional models to be incorporated into Fluidity by users. However, these models execute slowly due to their execution inside the Python interpreter. Figure 2.7 gives an example of a model that re-normalises a vector field using the Python interface. The Python interface has been used to implement four-component and six-

```
from numpy.linalg import norm
u = state.vector_fields['velocity']
for n in range(u.node_count):
    v = u.node_vals(n)
    m = norm(v)
    v[0] /= m
    v[1] /= m
    v[2] /= m
```

Figure 2.7: Re-normalising a vector field using the Fluidity Python interface. Fields are extracted from the `state` object dicts of scalar, vector, and tensor fields. Modification of field values in Python directly modifies the underlying arrays.

component biology models. However, it is more common to implement a new model in Fluidity using its Fortran API due to its efficiency compared with the Python interface.

Fluidity simulations are configured using Diamond, a GUI tool that automatically generates an interface for configuring all available parameters based on the options present in an XML file [HFG+09]. Figure 2.8 shows an example GUI window. The GUI allows users to set field values, for example, initial conditions, by writing a Python function that evaluates a field value at a given point. These functions are called at an appropriate time by Fluidity. The configurations that a user generates using Diamond are referred to as an *options file*.

2.6 Contemporary Parallel Architectures

The commonly-used architectures in scientific computing can be categorised as multi-core or many-core architectures. We characterise the fundamental difference between multi-core and many-core architectures as follows:

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

Many-core processors do not often operate independently of the main processor of a machine. The CPU of a machine is responsible for transferring input data into the

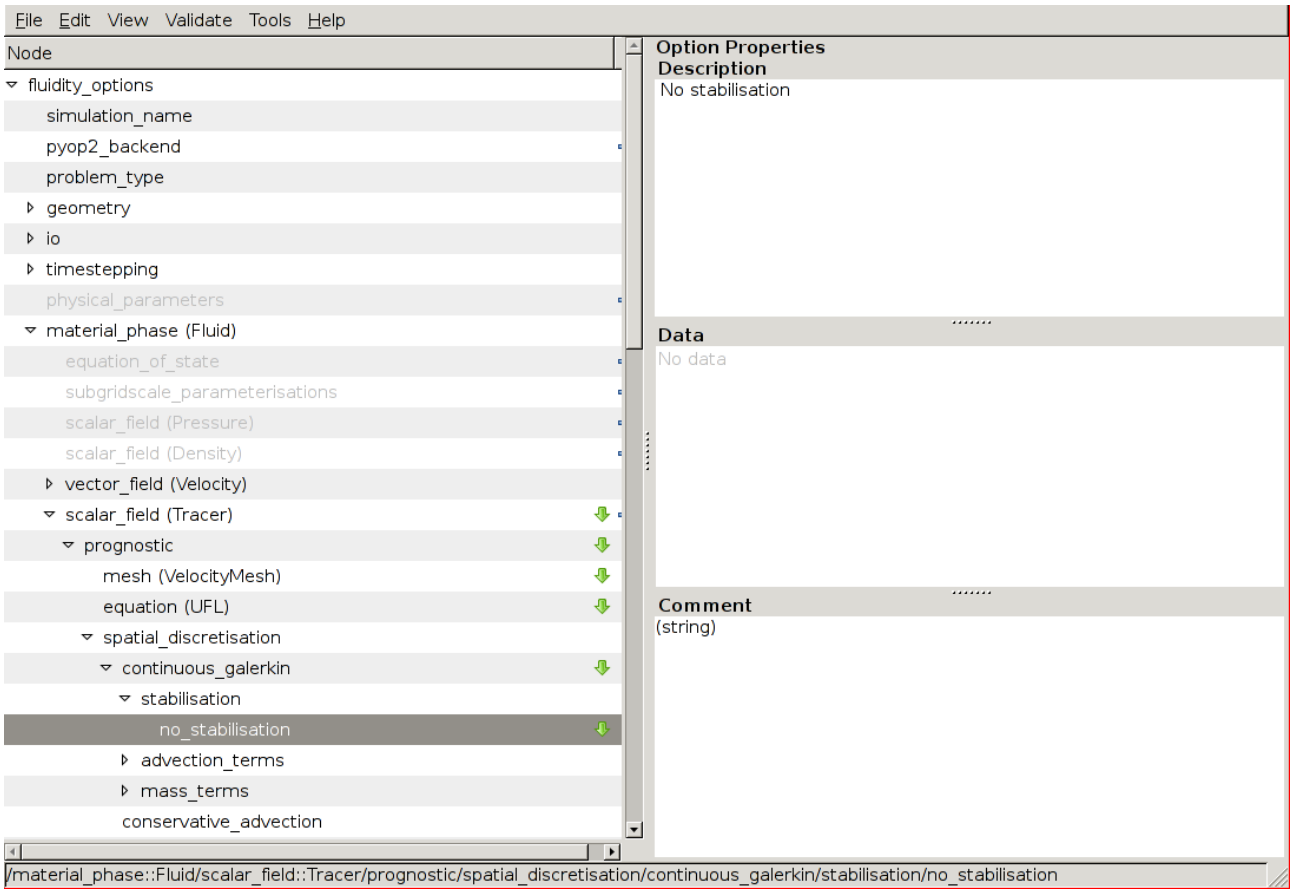


Figure 2.8: The Diamond GUI. The left pane contains the automatically generated options tree. The right pane is used to set options for a given node of the options tree. It is common to enter Python code here for setting boundary and initial conditions. Each node in the options tree has a path, shown in the status bar at the bottom of the window.

memory of a many-core device, launching *kernels* on the many-core device, and fetching the output data once computation is complete.

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

It could be argued that there is a general trend towards the convergence of these architectures. However, for the purposes of discussion and implementation on current platforms it is useful to consider them distinct designs.

2.6.1 NVidia Architectures

NVidia's Tesla, Fermi and Kepler architectures [LNOM08, NVi10a, NVi13] are representative instances of the many-core design. They are highly parallel architectures made up of many minimally complex processing elements which are specialised to perform arithmetic operations. The programming challenge on these architectures is the requirement to maintain thousands of in-flight parallel threads, each of which is restricted to a small number of registers and cache for maintaining its local state.

We will illustrate the architectures in more depth by describing the Fermi architecture. The Kepler and Tesla architectures differ from Fermi in the number of processing elements, maximum levels of concurrency, cache sizes and memory bandwidth, but the designs are similar in principle. Figure 2.9 gives an overview of the Fermi architecture. The processor consists of up to 16 *streaming multiprocessors* (SMs), all sharing a common level 2 cache and interface to main memory.

The architecture of an SM is outlined in Figure 2.10. The SM consists of 32 *streaming processors* (SPs). Each SP executes instructions on integer or floating-point operands. Two SPs are required for computation in double-precision, effectively halving the maximum throughput. The streaming processors also share a register file to support the execution of a large number of concurrent threads.

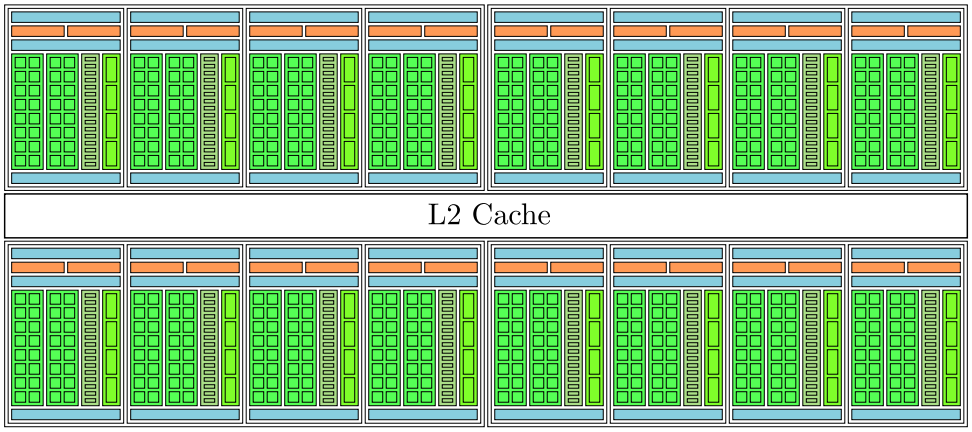


Figure 2.9: The Fermi Architecture [NVi10a]. Up to 16 Streaming Multiprocessors are on die, sharing a common L2 cache and interface to main memory.

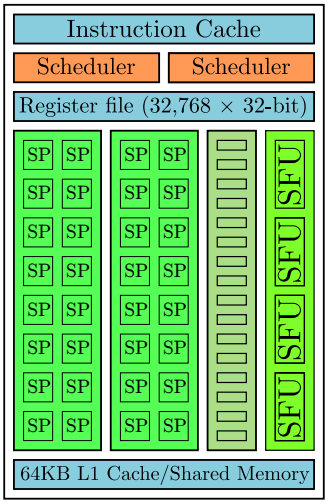


Figure 2.10: A single Streaming Multiprocessor [NVi10a]. 32 Streaming processors share a register file and shared memory. Four special function units (not considered here) compute transcendental functions at a reduced accuracy.

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

From the programmer's point of view, *kernels* are invoked on a Fermi processor under the control of the CPU. A kernel is a single function call executed on the many-core device in parallel by all of the cores. The programming model for kernels allows work (usually data-parallel operations) to be divided between a large number of *threads*. In NVidia architectures, threads are grouped at several granularities. A *warp* is a group of 32 threads that all share the same program counter, and as a result must all execute the same instruction concurrently. The next level of granularity is the *block*, which is made up of several warps. Each block is mapped onto a particular SM, and has an affinity to that SM for the lifetime of the kernel. No communication between threads in different blocks is allowed to take place. During the execution of a single kernel, one *grid* exists, which contains all the blocks. This organisation of threads and the lack of communication between blocks determines the strategies that are used to obtain good performance on a many-core architecture, which we describe in Section 2.6.3.

2.6.2 The AMD Evergreen architecture

Another instance of a many-core architecture is the AMD Evergreen architecture [Dev09]. Although AMD has subsequently produced newer architectures, we provide a description of Evergreen since it is used in our experiments in Chapter 4. The Evergreen hardware shares many design choices with the NVidia architectures. In particular, it consists of many simple cores. However, the arrangement of cores differs in Evergreen. The Evergreen processor consists of up 20 *SIMD engines*, each containing 16 *stream cores* (see Figure 2.11). Each of the stream cores consists of four processing elements that perform general purpose arithmetic, and one special function unit. Each machine instruction consists of an operation to be performed by

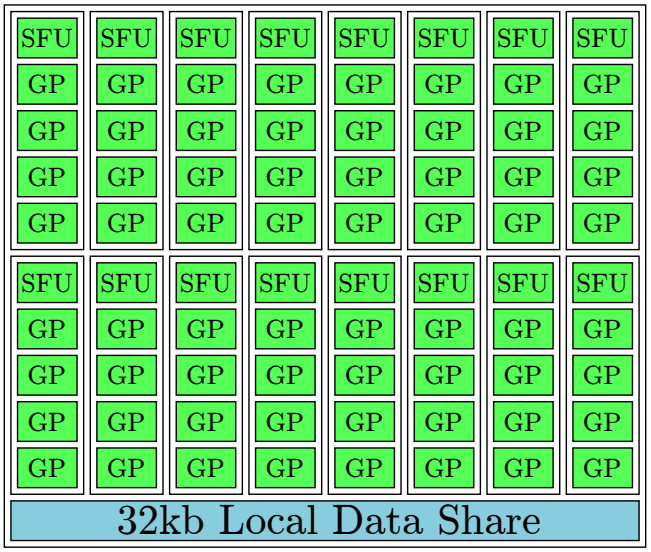


Figure 2.11: A SIMD Engine [Dev09]. Sixteen stream cores each with four general purpose processing elements and one special function unit, all sharing a 32KB LDS.

each of these units.

There is a 32KB software controlled cache on each of the SIMD Engines, which is referred to as the *Local Data Share (LDS)*. This performs a similar function to the shared memory in Fermi. Additionally, the SIMD Engines share a 64KB *Global Data Share (GDS)*, which is accessible by all of the SIMD Engines.

As with Fermi, individual kernels are launched on the device. The analogous concept of a warp on Evergreen is a *wavefront*, a group of 64 threads that are all executed in lock-step. Each wavefront is dispatched to an individual SIMD engine. Since a SIMD engine only contains 16 stream cores, each stream core executes each instruction for 4 threads sequentially. Although it is not possible to perform synchronisation between threads in different wavefronts, it is possible to implement global reductions across wavefronts by placing the reduced data into the GDS.

Since processing elements consist of several function units, the highest computational throughput can only be achieved when individual instructions operate on short (2- or 4-long) vectors. In simple cases the compiler can automatically generate instructions using these vector types, but in more complex ones, the programmer must explicitly write code to use vector data types.

2.6.3 Performance Considerations

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

- A massive amount of fine-grained parallelism is required in order to make full use of all the processing elements. This is typically rendered to the user as a requirement to launch thousands of concurrent threads.

- Because many threads run concurrently on a single multiprocessor, the per-thread state must be extremely small as it is stored within registers and a very small amount of cache. More registers per thread reduces the maximum number of running threads, reducing the scope to hide memory access latency. Additionally, the limited cache resources make the exploitation of temporal locality difficult.

- Spatial data locality is required across the threads in a warp, as *coalesced* memory access is needed for high bandwidth utilisation. Coalescing is achieved when vectors of threads concurrently access data within a certain-sized (typically 64 or 128 bytes) memory window.

- Performance is also dependent on spatial branch locality across the threads in a warp. Because warps all share a single program counter, they execute the same code path concurrently. When threads within a warp take different paths, execution is serialised between these two paths, reducing performance.

- Because many-core device memory is often separate from the main memory of a machine, data must be transferred to and from it before and after execution. Because of this overhead, it is important that a large enough workload is provided in order to benefit from the use of a many-core device.

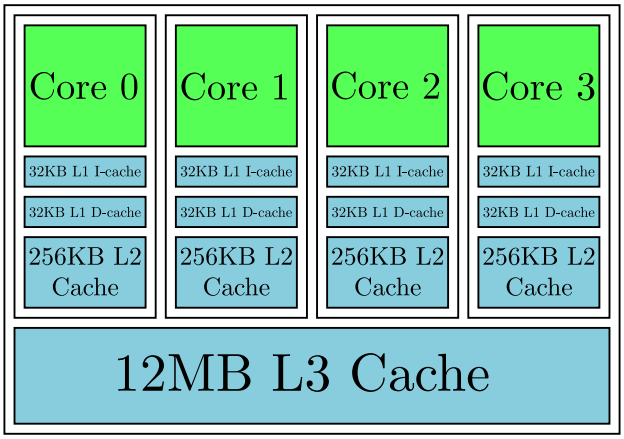


Figure 2.12: The Nehalem architecture. A small number of complex cores have large L2 caches and share a very large L3 cache.

2.6.4 Multi-core architectures

Since multi-core architectures are very similar to the single-core CPU architectures that are ubiquitous in high-performance computing, we avoid giving an exhaustive description of their design. Instead, we contrast the design of multi-core architectures with that of many-core architectures, and briefly describe how these differences imply a different strategy for writing efficient code. Figure 2.12 gives a schematic overview of Intel's Nehalem architecture, which is a recent instance of a multi-core architecture. The key differences in multi-core architectures include:

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

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

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

Instead of decomposing the workload into thousands of small data-parallel tasks, the usual implementation strategy on a multi-core architecture is to partition the workload in a coarse-grained manner. For example in finite element assembly, one may choose to partition the mesh

into several regions, assigning one region to each core. Each core would perform assembly for its region independently. By contrast, it would be more usual on a many-core architecture to decompose the domain into individual elements, and assigning one element to each thread.

2.7 Parallel Programming for Multi- and Many-core Architectures

In this section we provide a brief overview of low-level languages for programming many-core devices. We also discuss Æcute and OP2, which are higher-level abstractions for generating code with efficient data movement on multi- and many-core architectures.

2.7.1 CUDA and OpenCL

In order to make effective use of many-core architectures, programs must be designed such that the workload is decomposed into many (thousands) of data-parallel tasks that can be mapped to individual threads.

CUDA [NVi12a] is a language for programming NVidia devices. It is a set of extensions to the C programming language that allows the user to define kernels for execution on the device, and includes additional keywords that are used for managing the execution of threads.

In order to give a brief overview of a CUDA kernel, we consider an example of a function that performs the computation $\mathbf{y} = \alpha\mathbf{x} + \mathbf{y}$ for scalar α and vectors $\mathbf{x}$ and $\mathbf{y}$. A C implementation of this operation that uses a loop to iterate over each element in the vectors is shown in Figure 2.13.

```
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 2.13: DAXPY Kernel in C.

In order to convert this to a CUDA kernel (Figure 2.14), the work is divided between threads so that one thread computes the result for each element. This kernel is designed to be launched with as many threads as there are vector elements. Each thread calculates the offset for the element that it is assigned from the ID of its thread block, and its own ID within the block.

```
__global__ void daxpy(double a, double *x, double *y, int n) {
  int i=threadIdx.x+blockIdx.x*blockDim.x;
  if(i<n)
    y[i] = y[i] + a*x[i];
}
```

Figure 2.14: DAXPY Kernel in CUDA.

In order to standardise development for multi- and many-core architectures, the OpenCL specification [Mun11] has been developed. The OpenCL language shares many of the features of CUDA. However, it is designed to be compiled to executable code for a wide range of architectures, including multi-core, many-core and embedded processors. Since it is designed to be more flexible than CUDA in terms of supported targets, the assumptions about grids and blocks that are part of CUDA are more abstract in OpenCL kernel code. Figure 2.15 gives an example of the `daxpy` kernel in OpenCL.

```
__kernel void daxpy(const double a, __global const double *x,
                    __global double* y, int n) {
    int i = get_global_id(0);
    if (i >= n)
        return;

    y[i] = y[i] + a*x[i];
}
```

Figure 2.15: DAXPY Kernel in OpenCL.

Although it is relatively easy to translate existing codes to CUDA and/or OpenCL such that the execution produces correct results, optimising the performance of these codes is non-trivial. For example, it is often the case that the optimal division of work between threads and the granularity of kernels is not obvious, and various experiments must be performed in search of an optimum. Each time a new architecture is targeted, existing codes must be re-optimised, requiring a large investment of time and effort.

Additionally, managing the marshalling of data to and from accelerator devices, and between

the various levels of the memory hierarchy places further burden on the programmer. For example, making full use of the shared memory on Fermi or the LDS on Evergreen requires code to be written that explicitly marshals data at the beginning and end of execution of each kernel. Since the characteristics of the cache vary between architectures, this adds another dimension of complexity in maintaining code for multiple targets.

2.7.2 Æcute

Æcute [HLDK09] provides a programming model where the specification of the operations performed by a computation are decoupled from the specification of the data that it accesses. This decoupling is motivated by the fact that the performance optimisation of codes for multi-core systems and accelerators consists mainly of optimising data movement. Optimising data movement is difficult and error-prone, and inhibits performance-portability since the source becomes tuned to a specific architecture. Decoupling the access and execute specifications allows the automatic generation of data movement code, allowing the programmer to specify an algorithm in a clean and portable manner. The definitions and example that are presented in this section are developed in [HLDK09].

A kernel has an Æcute specification, which is a tuple, $S = (A, E)$, consisting of *access* metadata, A, and *execute* metadata, E. Execute metadata describes the iteration space that the kernel executes over. It is a tuple, $E = (I, R, P)$, where:

- $I \subset \mathbb{Z}^n$ is a finite, n-dimensional iteration space, for some $n > 0$.
- $R \subseteq I \times I$ is a precedence relation such that $(i_1, i_2) \in \mathbb{R}$ iff iteration i_1 must be executed before iteration i_2.
- P is a partition of I into a set of non-empty, disjoint iteration subspaces.

The execute metadata describes a polyhedral iteration space and allows dependencies between iterations to be specified in the relation R. The kernel may be applied to elements of the

iteration space in any order that meets the constraints specified in R. The purpose of the partitioning P is to generate subsets of the iteration space that can be loaded into a level of the memory hierarchy so that data in the subset can be cached and reused efficiently.

The access metadata describes the data that an individual iteration of the kernel will access. It is a tuple, $A = (M_r, M_w)$ where

- $M_r : I \to \mathcal{P}(M)$ specifies the set of memory locations $M_r(i)$ that may be read on iteration $i \in I$.
- $M_w : I \to \mathcal{P}(M)$ specifies the set of memory locations $M_w(i)$ that may be written on iteration $i \in I$.

The set of memory locations accessed on a given iteration is often a function of the *iteration vector*, i - in this case the set of memory locations accessed depends on the value of the current iteration.

Example: Matrix-Vector Multiplication

A matrix-vector multiplication $y = Ax$ can be implemented using a two-dimensional iteration space which corresponds to the i and j indices of the matrix elements. Since the vectors x and y are one-dimensional, the iteration space is projected to obtain the indices into the vectors.

For a $W \times H$ matrix partitioned into blocks of size $w \times h$, an Ælcute specification is $S = ((M_r, M_w), (I, R, P))$ where:

- $I = \{(i, j) : 0 \leq i < H, 0 \leq j < W\}$
- $R = \{((i, j), (i, k)) : 0 \leq i < H, 0 \leq j < k < W\}$
- $M_r(i, j) = \{A[i][j], x[j]\}$
- $M_w(i, j) = \{y[i]\}$

- $P = \{\{(i,j) \in I : h(k-1) \leq i < hk, w(l-1) \leq j < wl\} : 1 \leq k < H/h, 1 \leq l < W/w\}$

The relation R specifies that the loop indexed by i can be executed in parallel whereas the loop indexed by j is serialised. This constraint is not inherent in the matrix-vector multiplication algorithm, but illustrates how the choice of precedence relation can govern the performance of the generated implementation. If an implementation that serialises the loop indexed by j performs sub-optimally (for example if it causes data to be accessed with a large stride) then the precedence relation may need to be changed in order to generate a more performant version. In the matrix multiplication example, changing the relation such that the loop over i is serialised may improve performance. Since any affine relation that does not violate the data dependencies between iterations can be used, there is scope for generating many implementations by only varying the precedence relation.

Remarks

A prototype implementation of the Æcute paradigm is developed in [HLDK09] as a C++ template library for the Cell BE processor [Hof05]. Performance similar to that of hand-coded optimised implementations of several benchmarks (including a close-to-mean filter, a matrix-vector multiply, and a bit-reversed data copy) is obtained from this framework, and the generated implementations significantly outperform a software-based cache implementation.

The implementation of an Æcute framework for GPU-like devices has also received some consideration [HLKD09, How10]. One of the challenges in building efficient GPU implementations involves finding a suitable padding and alignment for data that maximises performance. It is shown that a poor choice of padding, or iteration ordering, can have severe impacts on the performance of the overall code. However, Æcute specifications permit the generation of algorithms with variations in padding and iteration order by modifying the Æcute specification, which is a relatively simple change compared to making changes to hand-written low-level data movement code.

The Æcute paradigm is well-suited to the efficient implementation of dense algorithms including

linear algebra and structured mesh PDE solvers. However, the affine nature of the maps M_r and M_w prevents the easy implementation of unstructured codes such as sparse linear algebra operations, and unstructured mesh PDE solvers. Although the Æcute model conceptually allows more general maps, the previously-published work has assumed that these maps are affine. OP2, which we describe next, can be seen as an extension of the Æcute model that permits non-affine maps.

2.7.3 OP2

OP2 is the second iteration of the Oxford Parallel Library for Unstructured Solvers (OPlus) [BCG94]. Although OPlus pre-dates Æcute, it can be thought of as a generalisation of the Æcute paradigm to unstructured iteration spaces. The designs of OPlus and OP2 are heavily motivated towards unstructured mesh CFD applications. The OPlus implementation was designed as a library for implementing distributed-memory parallelism using MPI, and we do not consider it further in this thesis. OP2 has been designed as an active library [VG98] primarily for implementing unstructured mesh solvers on multi- and many-core architectures. The present implementation of OP2 provides backends for CUDA and OpenMP, and also supports use of MPI for distributed-memory parallelism.

We continue this section by describing the API provided by OP2 for the programmer, which is described in [MGS+12]. We will then consider an example of an OP2 program, and discuss the performance optimisation and code generation strategies used by OP2.

The OP2 API

The data model used by OP2 allows the programmer to define unstructured mesh entities using the following primitives:

Sets. Individual entities of the mesh (such as nodes, edges, or cells, etc) are declared as *Sets*. A Set is specified by its size and a name. There is no class hierarchy provided by OP2 for different types of entities - they are all treated similarly by the OP2 implementation.

Data on sets. *Dats* allow items of data to be associated with each element of a set. Data may be multi-dimensional, and any number of Dats can be associated with a Set. For example, the coefficients of a field in the finite element method are stored in a Dat. Multiple Dats are used to represent different fields. For example, pressure, velocity, and salinity may be represented by different Dats.

Mappings between Sets. *Maps* define relations between elements in two Sets, in order to describe the connectivity of the mesh. For example, a mapping between cells and vertices may be defined. Maps are used in OP2 to express access descriptors in the same way as M_r and M_w in Ækute.

The execution model of OP2 allows the user to invoke *kernels* that execute for every element of a given Set in parallel. These parallel loops may execute over the set in any order, since unlike in Ækute, there is no ordering relation between iterations. The access specification for the kernel is given as a list of Dats that are accessed either *directly* or *indirectly*:

Direct access When a Dat is specified for direct access, the data from the Dat that corresponds with the Set element for the current iteration is passed to the kernel. No data for any neighbouring set element can be accessed.

Indirect access When a Dat is specified for indirect access, it allows data for neighbouring elements to the current Set element, or data defined on another Set to be accessed. A Map must be specified for indexing the neighbouring set elements. The present OP2 implementation only allows a single Map to be passed, so only one level of indirection is possible. It is possible to extend the implementation to support arbitrary levels of indirection through multiple Maps. However, indirection makes it very difficult to achieve the required coalescing on many-core architectures, so the use of multiple levels of indirection would be detrimental to performance. Most kernels in unstructured mesh CFD applications (for example, finite element assembly) can be implemented using a single level of indirection. In the case where multiple levels of indirection are required, one may

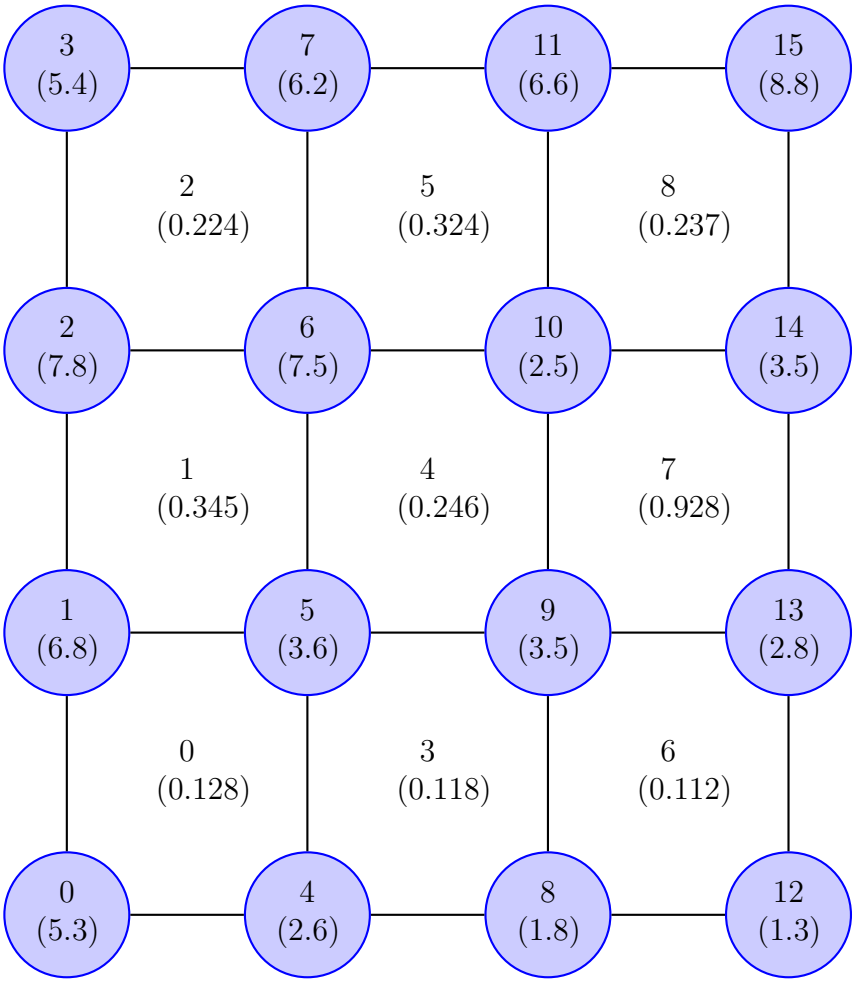


Figure 2.16: An example of a mesh, used in our sample OP2 program. Vertices (the circles) have data defined on them, as do cells (enclosed by four vertices).

compute the composition of the required Maps that can then be passed to OP2 as a single Map.

Each Dat is also specified along with an access specifier, which is either Read, Write, Read/Write or a reduction (Inc, Max, or Min). The specifier is used by OP2 to determine how conflicts should be avoided, or if special treatment is required to complete a reduction.

An Example OP2 program

The example mesh and code in this section are reproduced from [MGS+12]. We define a small mesh and a single kernel that is executed over this mesh in order to illustrate the OP2 abstraction, and its implementation in a C++ API. The kernel uses values associated with the

```
int nvertex = 16, ncell = 9;
op_set cells = op_decl_set(ncell, "set_cells");
op_set vertices = op_decl_set(nvertex, "set_vertices");
```

Figure 2.17: An example of the declaration of Sets of mesh entities in OP2.

```
int cell_map[36] = { 0,1,5,4,   1,2,6,5,    2,3,7,6,
                     4,5,9,8,   5,6,10,9,   6,7,11,10,
                     8,9,13,12, 9,10,14,13, 10,11,15,14};
op_map mcell = op_decl_map(cells, vertices, 4, cell_map,"cell_to_vertex_map");
```

Figure 2.18: An example of the declaration of Maps between Sets in OP2.

nodes and cells in order to compute an updated value in another field associated with the cells. Figure 2.16 shows a representation of the mesh used in this example. Note that although this mesh is structured, any unstructured mesh can be defined since the Map declarations provide the ability to specify arbitrary connectivity information.

The example consists of a mesh made up of cells that each have four vertices. Although the mesh clearly has edges, we do not make use of edge connectivity information in this example, so there is no need to explicitly define the edge information in the program. The declarations shown in Figure 2.17 specify the Sets used in the example.

The mapping from cells to vertices is required. This will be used to allow the kernel to access data at the vertices when iterating over cells. The mapping is first initialised as an array which is then passed to a function that constructs a Map, as shown in Figure 2.18.

Finally, we declare the data associated with cells and nodes. There are two fields on the cells: `dcells` and `dcells_u`. The field associated with the nodes is `dnodes`. Since `cell_data_u` is going to be overwritten, it is not initialised with any data. The declaration of data is shown in Figure 2.19.

We now declare a kernel that sums a cell value (`cell`) and the relevant node values (`n0-n3`) for the current cell. The summed value is stored back into another cell value (`cell_u`). Note that the kernel specifies the computation that is executed for a single set element. It is the responsibility of the OP2 runtime to pass in the correct data to the kernel. The kernel definition and the parallel loop invocation which informs OP2 of the data that is to be passed to the kernel is

```
double cell_data[9]  = {0.128, 0.345, 0.224, 0.118,
                        0.246, 0.324, 0.112, 0.928,
                        0.237};
double vertex_data[16] = {5.3, 6.8, 7.8, 5.4, 2.6, 3.6,
                          7.5, 6.2, 1.8, 3.9, 2.5, 6.6,
                          1.3, 2.8, 3.9, 8.8 };
double *cell_data_u  = (double *)malloc(sizeof(double)*9);

op_dat dcells    = op_decl_dat(cells, 1, "double",
                               cell_data, "data_on_cells");
op_dat dcells_u  = op_decl_dat(cells, 1, "double",
                               cell_data_u, "updated_data_on_cells");
op_dat dvertices = op_decl_set(vertices, 1, "double",
                              vertex_data, "data_on_vertices");
```

Figure 2.19: Declaration of data in our OP2 example.

```
void kernel(double* cell_u, double* cell,
            double* n0, double* n1, double* n2, double* n3){
    *cell_u = *cell + n0[0] + n1[0] + n2[0] + n3[0];
}

op_par_loop(kernel,"kernel", cells,
            op_arg(dcells_u,  -1, OP_ID, 1, "double", OP_WRITE),
            op_arg(dcells,    -1, OP_ID, 1, "double", OP_READ),
            op_arg(dvertices,  0, mcell, 1, "double", OP_READ),
            op_arg(dvertices,  1, mcell, 1, "double", OP_READ),
            op_arg(dvertices,  2, mcell, 1, "double", OP_READ),
            op_arg(dvertices,  3, mcell, 1, "double", OP_READ));
```

Figure 2.20: Example definition of a kernel and its invocation as a parallel loop in OP2.

```
op_par_loop(kernel, par_loop_name, it_set, arg1, arg2, ..., argN);

op_arg(dat, index, map, dimension, type, access);
```

Figure 2.21: The general form of a parallel loop in OP2. Multiple arguments are accepted, which are constructed by the `op_arg` function.

shown in Figure 2.20.

Kernel invocations in OP2 have the general form shown in Figure 2.21, where `kernel` is the name of the kernel to launch and `it_set` is the set that is iterated over, followed by one or more arguments. The `arg` parameters are constructed using the `op_arg` function, where `dat` specifies the name of the Dat to be passed. A mapping may also be passed if the Dat is to be accessed indirectly. If the Dat is to be accessed directly, then instead of passing a Map, `OP_ID` is used as a placeholder, and the index is set as -1. Since Maps may be multidimensional, the `index` parameter is used to indicate the index into the map that is used to indirectly access the Dat. There are four vertices per cell in our example, so the `dvertices` Dat is passed four times with the indices 0-3 into the mapping. This allows the kernel to use data from all four vertices. The requirement to enumerate all the indices of a Map through multiple arguments can be problematic: because of the resource limitations on many-core architectures, it is common that the space provided for kernel arguments is as little as 256 bytes, limiting the number of kernel arguments to 32. This number of arguments will already be exceeded by a Map from cells to degrees of freedom for tetrahedral elements with Lagrange basis functions of degree $p = 4$. Because this constraint is problematic for the implementation of finite element assembly, it is revisited in Section 6.2. The `dimension` and `type` arguments tell OP2 the dimension and type of the Dat - these must be consistent with the declaration of the Dat, and are required because OP2 is unable to perform analysis to determine the dimension and type of a Dat at the point of its use. Finally, `access` is the access descriptor, which is one of `OP_READ`, `OP_WRITE`, `OP_RW`, `OP_INC`, `OP_MAX`, or `OP_MIN`.

Parallel Execution Strategy

Considerations for parallel execution on a many-core platform include the staging of data into and out of the shared memory, and the avoidance of conflicting memory accesses when a parallel loop uses indirect accesses. In OP2, staging data into shared memory is facilitated by breaking the iteration set into small contiguously numbered *mini-partitions*. The size of the partitions is chosen such that the working set of a partition fits inside shared memory. In order to avoid conflicts, two levels of colouring are employed: between set elements within partitions, and between partitions.

A parallel execution *plan* is generated for each parallel loop in an OP2 program. The operation of the plan function is described fully in [Gil12b]. Here, we provide a brief overview of the stages in the plan generation, which are:

Renumbering. Mini-partitions consist of contiguously-numbered iteration set elements, but the mappings from these elements to indirect data are not contiguous. A renumbering of the mapping is generated to create a *local* mapping, that can be used to access indirect data once it has been staged into shared memory in a contiguous chunk of memory. This mapping is computed by generating a list of all the indices of the indirect dataset that are accessed by the mini-partition. The list is then sorted, and duplicates are removed. This prescribes the *local-to-global* mapping, which is used to load elements of the dataset into shared memory. If data is to be written out, this map is inverted to generate the *global-to-local* mapping.

Element colouring. In order to avoid conflicts between the threads executing on a mini-partition in parallel, the elements within the partition are coloured such that no two elements within the partition indirectly access the same dataset element.

Mini-partition colouring. Parallel execution of partitions may also cause conflicts if two partitions indirectly access the same dataset element. Mini-partitions are therefore coloured using a similar strategy to the element colouring.

Mini-partition mapping. [1] All of the mini-partitions of a single colour must be executed before execution can proceed on the next colour. However, by construction mini-partitions of the same colour form a disjoint subset of the iteration set. The mini-partition mapping holds a list of pointers to each mini-partition which is sorted such that all pointers to the same colour mini-partitions are stored contiguously. The offset of the beginning of each colour is stored, so that it is straightforward to find all the partitions of a given colour for iterating over them.

When a parallel loop is launched, the plan cache is searched to check if an execution plan for the current loop has already been generated. Because of the static nature of Sets and Maps, the plan for a parallel loop that accesses data on a particular list of Sets through a given list of Maps can be used for any subsequent parallel loop that accesses data on the same Sets through the same Maps. In the event of a plan cache miss, the plan function is called and the resulting plan is stored in the cache.

OP2 Code generation

The active library approach taken by OP2 depends on the execution of a preprocessor over the source code before it is compiled by the target compiler. Figure 2.22 details the compilation process in OP2. A source file (`app.cpp` in the figure) is first transformed by the OP2 preprocessor to produce a modified version (`app_op.cpp`) in which the `op_par_loop` calls are removed and replaced with calls to generated functions. Each `op_par_loop` prompts the generation of a source file containing generated code, (`app_k1.cpp`, `app_k2.cpp` in the figure, where kN is replaced with the kernel name). All of these generated kernel files are included into `app_kernels.cpp`. `app_op.cpp` and `app_kernels.cpp` are then compiled by the target device compiler and linked with external libraries as normal.

Each generated kernel file consists of a stub function that controls the invocation of a generated kernel. The stub function loops over the mini-partition colouring and invokes the device kernel

[1] This is referred to as *block mapping* in the OP2 developer documentation, which sometimes uses the term *block* to refer to a mini-partition. In this thesis, block is not used to refer to mini-partitions so as to avoid confusion between OP2 and CUDA terminology.

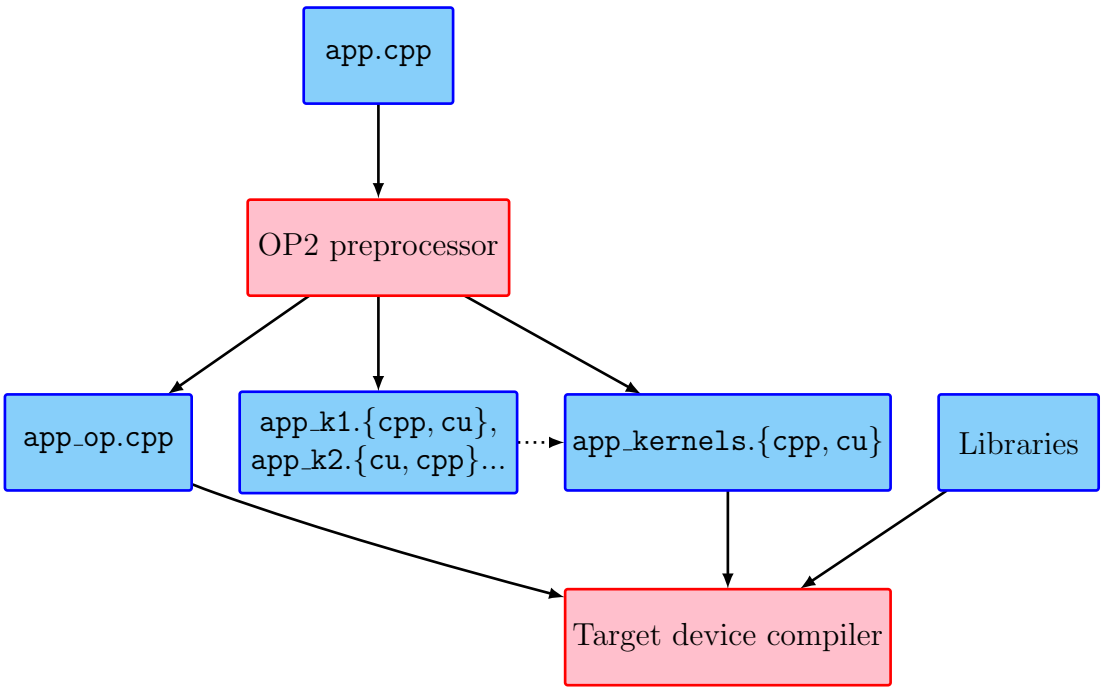


Figure 2.22: The OP2 build process (from [Gil12a]).

for each colour. The device kernel loops over the mini-partitions, and loads the data into shared memory for the mini-partition. In the device kernel, the intra-partition colours are then looped over, and the user's kernel is invoked for every element of that colour. Once all the partition colours have been executed, the data is written back from shared memory into device global memory.

Remarks

Performance studies using an airfoil test case that represents a structured mesh using OP2's unstructured representation have demonstrated high bandwidth utilisation on GPU devices [GMS+11]. The OpenMP and CUDA backends were examined, and the CUDA implementation showed a speedup over the OpenMP implementation commensurate with the difference in the target devices' peak performance.

The Airfoil test case serves as a reduced model of the HYDRA CFD code. HYDRA is a computational fluid dynamics package written in Fortran using the OPlus library. OP2 is also being used to port HYDRA to use multi- and many-core architectures [BBL+12]. This demonstrates that OP2 can be used with large, legacy Fortran codebases as well as C source

```
template <class T, int n>
void zero(T v[n]) { for (int i=0; i<n; ++i) v[i] = 0; }

template <class T, int n>
void zero_dat(op_dat d) {
  op_par_loop(zero, "zero", d.set(),
              op_arg(d, -1, OP_ID, n, T, OP_WRITE);
}
```

Figure 2.23: An attempt at writing a generic OP2 kernel for setting all entries of a Dat to 0. This attempt fails to be useful since the dimension of the Dat `n` often cannot be predetermined at compile time.

code. The Volna shallow-water modelling code [DPD11] has also been ported from C to use OP2.

The partitioning algorithm used in the parallel algorithm strategy creates consecutively-numbered blocks. Although this algorithm is straightforward, it can cause suboptimal performance with poorly-ordered meshes - if consecutively numbered elements are not proximate, then there will be little data reuse between elements in a partition. This is discussed further in Chapter 7.

The code transformation process used by OP2 operates in a straightforward manner, but this causes some difficulty for the engineering of large, complex codebases. In particular, the OP2 primitives are not integrated with the type system of the host C++ language. This means that the OP2 translator is unable to perform analysis to determine the type of datasets and their dimension at the point of use; hence, these attributes are always specified by the programmer in an `op_arg` declaration.

An additional issue with the static compilation approach is that the code generation for parallel loops is specialised to the dimension and type of the Dats used in them. This prevents the definition of general forms of parallel loops. As an example, a useful generic function could be one that initialises a Dat to a zero value. We would perhaps like to write it as shown in Figure 2.23. This would allow the instantiation of the correct versions of the kernel if the type could be determined statically. However, the type cannot be determined at compile time in many cases. For example, if a conditional is used to control the assignment of a Dat, as shown in Figure 2.24, the version of the kernel to invoke is undetermined at compile time. Consequently, we see that it is not possible to write generic functions in this manner in OP2.

```
op_dat d;
if (cond()) d = dat1;
else        d = dat2;
zero_dat(d);
```

Figure 2.24: Invocation of different versions of the `zero_dat` kernel based on runtime information. This does not work in OP2, as control flow must be resolved at compile time.

2.8 Conclusion

Our brief overview of the finite element method has been used as a basis for the description of the Unified Form Language, and how it is used in the development of solvers implemented using FEniCS/DOLFIN. We have discussed multi-and many-core architectures, and the challenges in obtaining good performance when programming in the low-level languages that are commonly used to develop software for them. Finally, we have discussed approaches that decouple the specification of computations from the specification of their accesses, which automate the generation of data movement code.

The material covered in this chapter serves as a basis for the developments and experiments presented in Chapters 4-7. The following chapter will discuss related efforts that attempt to bring performance portability and maintainability to scientific computing, and the use of code generation to achieve these goals.

Chapter 3

Performance-portability in Scientific Applications

3.1 Introduction

In this chapter we explore recent efforts to develop maintainable, robust and performance-portable scientific codes. We examine implementations of finite element methods on many-core architectures, and performance optimisation strategies for the finite element method on multi- and many-core architectures. We also discuss code generation techniques and programming tools that facilitate the development of finite element codes. Finally, we examine approaches in other areas of scientific computing that use domain-specific languages and code generation tools to enable performance-portability, or to reduce the level of complexity that developers need to manage.

3.2 Finite Element Methods on Multi- and Many-core Architectures

We focus mainly on many-core architectures in this section. However, we also discuss an implementation on multi-core architectures in Section 3.2.5 because it highlights important tradeoffs that can be made in finite element implementations.

3.2.1 Global Assembly Strategies on GPUs

An exploration of the implementation space for global assembly on GPUs is presented in [CLD11]. The experimental investigation explored the use of different levels of the memory hierarchy for storing local matrices prior to their addition into the global matrix, and varying the granularity at which the workload is assigned to threads. Four strategies are benchmarked:

1. Performing local assembly and global assembly using one thread per element. No intermediate storage of local matrices is used, but local matrix terms are added directly into the global matrix after they are computed. Element colouring is used to avoid conflicts as atomic operations were not supported by all the platforms used in the benchmarks.

2. Performing local and global assembly using one thread per non-zero term. Local matrices are written out to global memory by the local assembly kernel. The global assembly kernel then reads these values from global memory before reducing them. Since one thread is assigned to each non-zero in the global matrix, there is no contention.

3. Use of shared memory for storing intermediate results. In this strategy, small partitions of the mesh are loaded into shared memory so that element data can be reused. A thread block cooperates to compute the local matrices for the partition. Subsequently, one thread per global matrix non-zero adds terms into the global matrix. In order to avoid conflicts at partition boundaries, the degrees of freedom on a boundary are only owned by one partition, and only the owner adds terms into the global matrix. This requires a small amount of redundant data transfer and computation.

4. Use of local memory (i.e. registers) for the storage of intermediate results. One thread is assigned to each global matrix entry, and computes the contributions to it from all the associated elements. This strategy requires a large amount of redundant computation and data transfer, since values cannot be re-used between terms that would otherwise have belonged to the same local matrix.

Experiments are performed using a CUDA implementation of a 2D heat equation solver run on the NVidia 8800 and Tesla C1060. All computations used single-precision arithmetic as double-precision was not supported by the NVidia 8800. This is in contrast to the double-precision arithmetic used in the experimental work described later in this thesis. The conclusions drawn consist of two cases:

- When performing assembly using low-order finite elements (generally where $p < 4$), it is preferable to use Strategy 3, where shared memory is used for element data and intermediate results. This strategy works well for low-order elements because the partition size and the number of local matrices that can be held in shared memory are inversely proportional to the element order. Once partitions become small when using high-order elements, there is little opportunity for reusing values that have to be loaded into shared memory.

- When assembling higher-order elements, it is preferable to use Strategy 2, where local matrices are written back to global memory before being assembled by a subsequent kernel. Although this strategy requires more data transfer than Strategies 1 and 4, it uses a smaller number of floating-point operations, which can become intensive for higher-order elements.

In our implementation described in Chapter 4, an assignment of one thread per element of the local matrix is used for the global assembly strategy. Note that although this is contrary to the recommendations for global assembly in [CLD11], it allowed for coalesced access with the mesh data structures that were chosen.

3.2.2 A CUDA and MPI SPECFEM3D Implementation

A port of the SPECFEM3D code for seismic simulations to CUDA and MPI is described in [KME09, KGEM10, KEGM10]. All computation takes place on the GPUs, whilst the CPUs are only used for exchanging data via MPI.

The simulations used hexahedral elements with $p = 4$, resulting in local matrices that have 125×125 entries. The size of these local matrices makes it practical to assign an entire thread block to the computation of a local matrix. Blocks of 128 threads were used and element data was padded up to 128 elements. Although the final three threads in each block were redundant, the padding simplified the alignment of data and assignment of threads to elements. A similar mapping of thread blocks to local matrices is enabled by the abstractions that we have designed for the PyOP2 framework described in Chapter 6.

In order to avoid conflicts when assembling global matrices, the mesh elements are coloured. The use of atomic operations to avoid conflicts was not investigated since NVidia GT200 and S1070 devices used for the experiments only supported atomic operations on integers.

Communication and computation is overlapped when possible in order to improve performance. This is achieved by launching a kernel that performs computations on halo elements first, then beginning a non-blocking halo exchange. Whilst halo exchanges are in-flight, computation is performed for the interior elements. This is similar to the strategy implemented for MPI in PyOP2, described in Section 6.5.5.

Near-perfect weak scaling up to 192 GPUs is reported. This can partly be attributed to the relatively large workload of the interior portion of partitions compared to the halo regions, which enables a large overlap between communication and computation. A reference CPU implementation also scales similarly, but it is difficult to draw a meaningful conclusion about the speedup of the GPU cluster over the CPU cluster due to the differences in the configuration of the machines.

3.2.3 Fusion of Primitive Kernels

As discussed in Section 2.6.3, threads executing on many-core architectures must maintain as small a state as possible in order to maximise parallelism. This mandates that the code size of individual kernels must be kept to a minimum. However, writing many small kernels increases global memory bandwidth requirements, since intermediate results that could normally be passed through registers or shared memory must be written out into a large array. It is therefore necessary to find a trade-off between using many small kernels and a few larger ones. Performing this task manually is undesirable since there will be many possible combinations in any non-trivial application, and the process will be error-prone. Furthermore, the optimal combination will differ between devices, so this optimisation process is not performance-portable.

In [FPF09a, FMFM13], a technique for automatically finding the optimal kernel size is discussed, with application to a finite element solver for hyperelastic equations, which can be used to model deformation in solid bodies. A set of kernels that implement all the primitive operations used in the assembly process are provided in an initial implementation. An automated translator fuses individual kernels together to generate new implementations. This enables a search of the implementation space in order to choose the optimal set of fusions. The optimal fusion implementation of the simulation described showed a speedup of approximately 100% over the baseline implementation.

Kernels that make different assignments of the workload to threads can still be fused. As an example, Figure 3.1 shows a sequence of four kernels. Kernels O_1 and O_3 assign 1 and 3 threads to each element respectively. In the fused version, the portion of the kernel from O_3 runs with two-thirds of the threads idle. Although this introduces inefficiency, the global memory transfer between O_1 and O_3 is eliminated, which can amortise the cost of leaving threads idle.

3.2.4 High-order Discontinuous Galerkin Methods on GPUs

An implementation of a discontinuous Galerkin solver for Maxwell's Equations is described in [KWBH09]. This implementation is generated using a domain-specific compiler for the

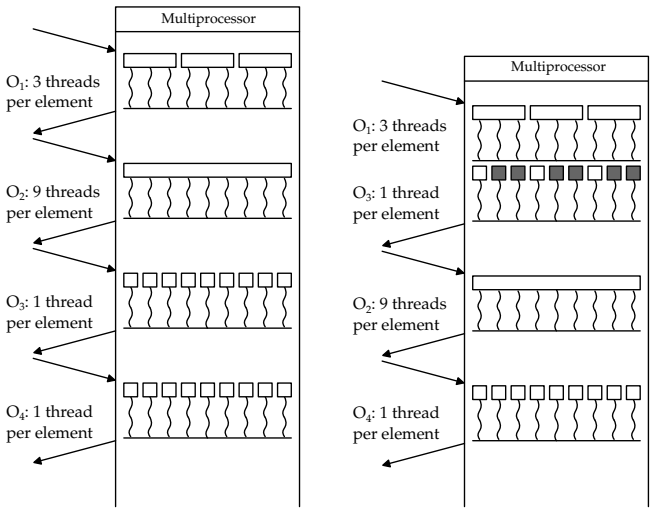


Figure 3.1: Fusing kernels of different granularities. *Left:* Unfused kernels store results in global memory. *Right:* One pair of kernels is fused. This fusion is possible when O_3 does not depend on the output of O_2. From [FPF09b].

implementation of discontinuous Galerkin methods in CUDA, called Hedge. The language constructs provided by Hedge allow the user to specify *operator templates*, which are expression trees representing the computation performed by an operator.

Data layout optimisation in this implementation involves partitioning the mesh into small chunks that can be co-operatively loaded into shared memory by a thread block, before computation proceeds on this data. Since data is reused between elements that share a face, it is more efficient to try to load in a set of elements that share as many faces as possible. This can be achieved by partitioning the mesh into many very small partitions that consist of the maximum number of elements that can be loaded into shared memory. It has been reported that standard partitioning tools (such as [KK98]) fail to work efficiently for creating such partitions, as they are designed with partitioning a mesh into fewer, larger sets in mind [KWBH09]. Instead, a simple greedy partitioning algorithm has been shown to be effective for this purpose.

The number of floating-point values needed to represent the data for a single element often does not equally divide 16, which is the number of values read in a single coalesced read. Instead of placing all element data contiguously in memory and performing a complicated calculation of the offset to access data, a padding scheme is used, which is illustrated in Figure 3.2. Element data that can fit within 64 bytes (16 float values) is packed contiguously. The remaining space up to 64 bytes is padded. This simplifies the staging of data into shared memory by cooperating

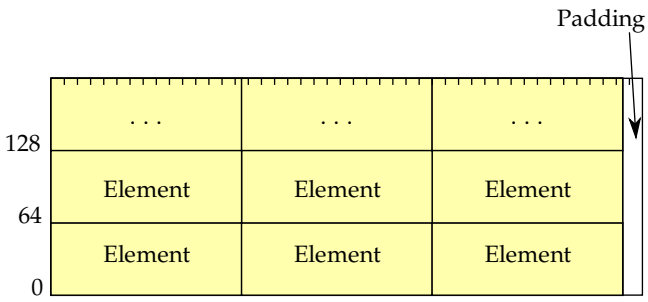


Figure 3.2: Data layout for threads to co-operatively load element data from a small partition into shared memory. From [KWBH09].

threads.

Most of the computations used in the discontinuous Galerkin method are element-local operations. These are the flux lifting, function evaluation and local differentiation components of the solver. The operation which is non-local is the *flux gather* stage, in which information is propagated between elements. Because the majority of operations are local, the matrix-free method is used for the evaluation of operators. In the matrix-free method, there is no local or global assembly process. Instead, when the action of a matrix operator is to be evaluated, the product of a local matrix and a vector of elemental values is computed directly by evaluating the expressions for the entries of the result vector in terms of the basis functions and other values that would be using in assembling a local matrix. The use of the matrix-free method in the DG implementation avoids uncoalesced memory accesses when performing the element-local operations.

3.2.5 Optimal h/p-discretisation

In [CSKK11b, CSKK11a, VSK10] the optimal choice of evaluation strategy for finite element operators is investigated. These strategies are:

- The *Global Matrix Approach*. Local elemental matrices are assembled into a global matrix which is used for the computation of a matrix vector product in the iterative solution process.

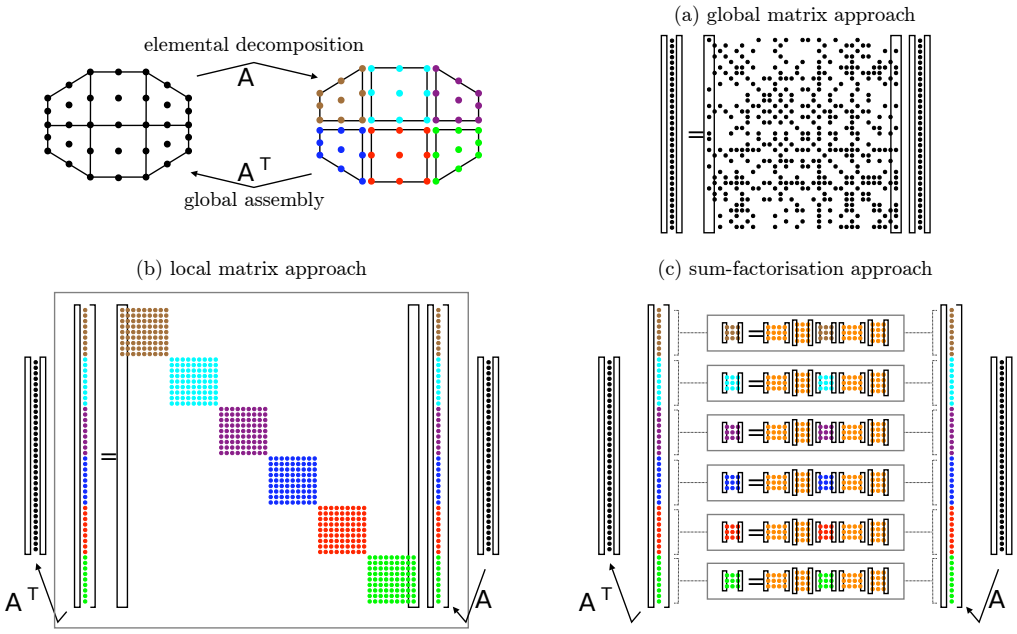


Figure 3.3: Schematic diagram of the operator evaluation strategies from [Vos11]. The operator $\mathcal{A}$ transforms a global representation to its element-local representation. This transformation is reversed by the operator $\mathcal{A}^T$. In the global matrix approach (a), these operators have been applied to construct the global matrix. In the local matrix and sum-factorisation approaches, the operator $\mathcal{A}$ is applied to a global vector, then $\mathcal{A}^T$ is applied to a resulting local vector.

- The *Local Matrix Approach*. Local matrices are assembled. The matrix-vector product is computed by mapping the multiplicand vector to a vector in the local elemental space, multiplying by local matrices, then mapping the result back into the global space.

- The *Sum-factorisation Approach*. This is similar to the local matrix approach but the tensor-product structure of the finite element basis is exploited to reduce the complexity of the local matrix evaluation.

We describe the local and global matrix approaches in detail in Chapter 4. A schematic diagram of the computations in each of these approaches is given in Figure 3.3.

The investigations show that the optimal algorithm depends on factors including the dimension of the problem, polynomial order of the basis functions, the equation being discretised, and the degree of continuity of the solution. For example, the best performance evaluating the Helmholtz operator on a quadrilateral mesh is obtained using the global approach at $p = 1 - 3$, the local approach at $p = 4 - 13$, and the sum-factorisation approach at $p \geq 14$. The choice of

optimal algorithm changes at different p values for other operators and discretisations.

These investigations provide the groundwork for making the correct choice of algorithm given a choice of h and p which are chosen to give a solution to a required accuracy. All experiments were conducted with implementations run on a multi-core processor. In Chapter 4, we also demonstrate that the optimal choice of algorithm changes at different p values on many-core architectures.

3.3 Program Generation for Finite Element Methods

In this section we examine code generation tools for implementing finite element solvers. In order to keep this section self-contained, we briefly reference some frameworks that are described in detail elsewhere and refer the reader to the relevant sections of this thesis.

3.3.1 FEniCS

We describe the FEniCS Project in detail in Section 2.4. There are a number of other finite element libraries and toolkits that inspired the development of the FEniCS toolchain [BS04, HPHO05, DG05, BHK07, Lan03, Lon03], which it has largely superseded. We do not consider these frameworks further. The FEniCS/DOLFIN Problem Solving Environment (PSE) [LW10] provides an environment for the rapid development of finite element solvers from within Python and has become widely used for the implementation and prototyping of finite element methods. Portions of the toolchain used by DOLFIN are used in the work presented in this thesis, in particular UFL and FFC are used in the software stack described in Chapter 7.

A novel transformation that was developed as part of the FEniCS project and implemented in the toolchain is the *tensor contraction representation*. This optimisation can greatly increase the performance of local assembly kernels by reducing the total operation count when evaluating integrals using Gaussian quadrature. We draw attention to the fact that although the transformation is complicated, as we will see, its implementation in FFC made it trivial to use

from the end user's point of view, since their finite element solver is implemented in UFL and the use of the optimisation is controlled by a commandline switch.

We now describe the application of the tensor contraction representation to a simple form. The optimisation works by allowing more of the computational work to be performed on the reference element. This reduces the work needed to transform the solution from the reference element to the physical element. Since the evaluation over the reference element is only performed once, whereas the transformation is performed for every element, this optimisation greatly reduces the cost of the overall computation.

We consider the transformation in the case when the mapping from the reference element to the physical element is affine. Consider a Laplacian form that is be evaluated using Gaussian quadrature as follows:

$$M_K^e[i,j] = \int_{\Omega^K} \nabla\phi_i \cdot \nabla\phi_j \, \mathrm{d}X = \sum_{q=1}^{N_q} \sum_{\alpha=1}^{N_{dim}} w_q \frac{\partial \phi_i}{\partial X_\alpha}(\xi_q) \frac{\partial \phi_j}{\partial X_\alpha}(\xi_q) |J^K|, \tag{3.1}$$

where w_q is a quadrature weight, J^K is the Jacobian of the transformation from reference space to physical space, and ϕ is a basis function in physical space. In order to derive the form that performs the transformation by a tensor contraction, we first write the dot product in the integral in terms of its components:

$$M_K^e[i,j] = \int_{\Omega^K} \nabla\phi_i \cdot \nabla\phi_j \, \mathrm{d}X = \int_{\Omega^K} \sum_{\alpha=1}^{N_{dim}} \frac{\partial \phi_i}{\partial X_\alpha} \frac{\partial \phi_j}{\partial X_\alpha} \, \mathrm{d}X. \tag{3.2}$$

Next, we can make a change of variables from the physical coordinates to the coordinates on the reference cell, giving:

$$M_K^e[i,j] = \int_{\Omega^K} \sum_{\alpha=1}^{N_{dim}} \sum_{\beta_1=1}^{N_{dim}} \frac{\partial \xi_{\beta_1}}{\partial X_\alpha} \frac{\partial \Phi_i}{\partial \xi_{\beta_1}} \sum_{\beta_2=1}^{N_{dim}} \frac{\partial \xi_{\beta_2}}{\partial X_\alpha} \frac{\partial \Phi_j}{\partial \xi_{\beta_2}} |J^K| \, \mathrm{d}X, \tag{3.3}$$

where ξ is a coordinate on the reference cell, and Φ is a basis function on the reference cell. Since the mapping from the reference element to the physical element is affine, the sums $\sum_\beta^{N_{dim}} \frac{\partial \xi}{\partial X}$

and the determinant $|J^K|$ are constant throughout the element, and can be moved outside the integral:

$$M_K^e[i,j] = |J^K| \sum_{\alpha=1}^{N_{dim}} \sum_{\beta_1=1}^{N_{dim}} \frac{\partial \xi_{\beta_1}}{\partial X_\alpha} \sum_{\beta_2=1}^{N_{dim}} \frac{\partial \xi_{\beta_2}}{\partial X_\alpha} \int_{\Omega^K} \frac{\partial \Phi_i}{\partial \xi_{\beta_1}} \frac{\partial \Phi_j}{\partial \xi_{\beta_2}} \, \mathrm{d}X. \tag{3.4}$$

This can be written as a tensor contraction:

$$M_K^e[i,j] = \sum_{\beta_1=1}^{N_{dim}} \sum_{\beta_2=1}^{N_{dim}} A_{ij\beta} G_K^\beta \qquad \Rightarrow \qquad M^e = A : G_K, \tag{3.5}$$

where

$$A_{ij\beta} = \int_{\Omega^K} \frac{\partial \Phi_i}{\partial \xi_{\beta_1}} \frac{\partial \Phi_j}{\partial \xi_{\beta_2}} \, \mathrm{d}X, \; G_K^\alpha = |J^K| \sum_{\alpha=1}^{N_{dim}} \frac{\partial \xi_{\beta_1}}{\partial X_\alpha} \frac{\partial \xi_{\beta_2}}{\partial X_\alpha}. \tag{3.6}$$

This optimisation reduces the number of arithmetic operations required from $N_q n_0^2 d$ to $d^3 + n_0^2 d^2 \sim n_0^2 d^2$ where N_q is the number of quadrature points, n_0 is the polynomial order of the basis, and d is the number of dimensions. Since the number of dimensions is almost always 2 or 3, the reduction in operations is roughly a factor of N_q, which can be a significant reduction for high-order basis functions.

We remark that although this particular example is specific to the Laplacian form, the key parts of this optimisation are the change of variables and movement of the constant terms outside the integral, which may be applied to any integral form. There are other examples of similar optimisations that may be used in cases when the transform from the reference element to the physical element is not affine, discussed in [KL06] and [KL12]. Note however that tensor contraction scales badly with the number of derivatives in the form.

3.3.2 OP2

OPlus2 [MGS+12, GMS+11] is a framework for writing parallel programs that perform computations on unstructured meshes, which we have discussed extensively in Section 2.7.3. It uses source-to-source translation to generate CUDA implementations of user-specified code. The OP2 C/Fortran implementation [MGS+12] shares many design decisions with the PyOP2 implementation outlined in Chapter 6. The main difference is that OP2 uses static code translation to generate the kernels whereas PyOP2 uses runtime code generation.

3.3.3 Nektar++

Nektar++ [VEB+11] is a C++ library for implementing finite element methods that allows one of several different assembly algorithms to be chosen. The key insight of experimental work done with Nektar++ is that the optimal choice of algorithm, even on the same machine, varies depending on the problem parameters [CSKK11a, CSKK11b, VSK10]. We note that this variation in the algorithmic choice also provides a motivation for generating code at runtime, in order to allow the specialisation of an implementation to the current problem parameters. Nektar++ currently targets only CPU architectures. However, it may be expected that the optimal implementation for a given set of parameters will vary with the target architecture.

3.3.4 Liszt

Liszt [DJP+11, DH11] is a DSL for unstructured mesh applications that is embedded in Scala. The goal of Liszt, which is to provide a framework for the development of performance-portable mesh-based PDE solvers, is very similar to that of OP2. On the surface, the programming interfaces provided by both Liszt and OP2 appear to be similar.

However, the design philosophy of Liszt comes from the opposite direction to OP2 - instead of providing the user with very simple abstractions from which to build a mesh representation, it prescribes a set of mesh entities with well-defined semantics. These entities include vertices,

cells, edges, and faces. The relationships between these mesh entities are described with semantic operators, such as the inside or outside of a cell. These semantic relations replace the concept of a Map in OP2.

The semantics of parallel loops are very similar to those in OP2. The execution within a parallel loop, a *phase*, is a data-parallel computation without dependencies. Outputs produced in one phase are only consumed in subsequent phases. These semantics permit a parallel Jacobi iteration but prevent a naive implementation of a parallel Gauss-Seidel iteration. However, the Gauss-Seidel algorithm may be implemented in Liszt using a Red-Black ordering.

Because there are no Map values provided in Liszt, program analysis techniques are used to compute the stencil of expressions within a phase. The analysis of expressions in a phase is used to guide its translation into a form in which unnecessary operations are eliminated.

Liszt has been used to implement finite volume solvers for the Euler, Navier-Stokes and Shallow Water equations, and a finite element solver for the Laplace equation. These applications were transformed to different implementations using Liszt's CUDA, pthreads and MPI backends. The generated implementations showed parallel speedups over a scalar implementation that were commensurate with the difference in performance between the parallel hardware and the sequential baseline.

The finite element solver for Laplace's equation is an implementation of a very simple equation. It is unclear whether the Liszt data structures are well-suited to implementing a variety of finite element problems, because the mesh abstractions that are provided do not appear to allow flexible specification of the degrees of freedom in a mesh. It is possible to use function spaces based on $p = 1$ Lagrange elements, but it may be difficult to use $p = 2$ and above basis functions, or more exotic spaces, such as those made from Raviart-Thomas or Brezzi-Douglas-Marini elements [KLRT12] due to this conflation of vertices and degrees of freedom.

The strong semantics of the Liszt abstractions provide a means to build PDE solvers using elegant representations of the mesh data structures. However, the abstract nature of these representations are further removed from the representations used in solvers developed in more

conventional languages such as C or Fortran. It therefore appears that it will be more complicated to bridge to existing implementations and toolchains than OP2.

3.3.5 Directive-based Approaches

The automatic porting of an OpenMP-parallelised Fortran code to run on multiple GPUs is described in [CCLM11, CL11]. This enables the conversion of all the loops over the mesh in a large finite element solver to run on GPUs. Since all of these loops are converted, the need to transfer data back and forth between the CPU is removed, as the entire computation can be performed on the GPUs.

Although this approach is well-suited to adapting existing software to make use of many-core architectures, annotating a source code with directives preserves its existing structure and algorithms. Whilst this minimises the effort required to adapt software, the use of annotations does not provide a code generation framework with enough high-level information about the algorithm or program structure to enable it to make optimising transformations. As a result, the annotated code is not necessarily performance-portable, so manual restructuring and tuning is still necessary.

3.3.6 EXCAFÉ

EXCAFÉ [Rus11] is an active library written in C++ that performs *expression capture* for generating optimised implementations of finite element solvers. It provides a mechanism for defining the entire set of operations that are used in a finite element computation. This gives the library a global view of all the operations in a method, which allows it to generate highly optimised code. The use of expression capture in EXCAFÉ is a progression of the use of expression capture to delay evaluation used in the DESOLA linear algebra library [RMKB11].

The interface for describing expressions in EXCAFÉ is sufficiently general that it can be used to develop non-trivial finite element solvers. Implementations of solvers for the advection-diffusion

equation and the Navier-Stokes equations have been developed. The expression capture also provides mechanisms for capturing timestepping, by permitting the annotation of expressions with a symbolic timestep. This allows the declarative specification of expressions for advancing the simulation time whilst allowing EXCAFÉ to infer the structure of the timestepping loop.

EXCAFÉ represents the local assembly matrix in terms of the polynomial expressions that are used to evaluate each of its entries. An extended version of Hosangadi's algorithm [HFK06] for the factorisation of polynomial expressions is used to simplify these expressions. This allows EXCAFÉ to generate local assembly code that is often more efficient than the standard quadrature representation and the tensor representation used in FEniCS [RK13]. The increase in efficiency is achieved by reducing the number of floating-point operations that are required to evaluate the local matrix, which directly translates to a reduction in execution time.

The manipulation of the symbolic representation of polynomial expressions by EXCAFÉ allows it to generate optimised local assembly kernels without loss of precision due to the finite nature of floating-point arithmetic. This is a further improvement upon the FEniCS tensor contraction representation, whose implementation loses precision for some local assembly kernels since it operates on the floating-point values of polynomial expressions rather than their symbolic representation.

EXCAFÉ differs chiefly from the Firedrake framework that we describe in Chapter 7 in that it provides a tightly-integrated implementation of all the components of a finite element toolchain, whereas the Firedrake framework is modular in comparison. Although this allows it to generate highly-efficient code, it will be relatively difficult to incorporate EXCAFÉ into an existing solver or finite element toolchain.

3.4 Scientific Application Frameworks

In this section we examine how domain-specific languages and code generation techniques have enabled performance-portability in other areas of scientific computing.

3.4.1 The Tensor Contraction Engine

The Tensor Contraction Engine [ABB+06, BBC+02] is used for searching for the most efficient implementation of a specified set of tensor contractions, given restrictions on the amount of storage space available at various levels of the memory hierarchy. We consider an example given in [BBC+02]:

$$S_{abij} = \sum_{cdefkl} A_{acik} \times B_{befl} \times C_{dfjk} \times D_{cdel} \tag{3.7}$$

This expression may be evaluated using ten nested loops in $4 \times N^{10}$ operations, where the range of each index is N. The same result can be computed in $6 \times N^6$ operations by rearranging the expression:

$$S_{abij} = \sum_{ck} \left(\sum_{df} \left(\sum_{ef} B_{befl} \times D_{cdel} \right) \times D_{dfjk} \right) \times A_{acik} \tag{3.8}$$

However, this new expression requires a large amount of temporary storage, which may exceed the memory capacity of the target machine. There is a large set of expressions that perform an equivalent computation to Equations 3.7 and 3.8, requiring varying amounts of re-computation and storage space. Manipulation of the tensor expression to produce various different versions is relatively straightforward, especially using a package such as Matlab [Mat09]. However, producing an implementation of the computation and benchmarking it requires a relatively large effort on the part of the programmer to implement the code needed to compute each expression.

In order to overcome this problem, the TCE provides a domain-specific language for specifying tensor expressions. An optimal implementation of an expression that requires a minimal amount of computation given memory and disk space limits is searched for by the engine, which can automatically generate an implementation. This automation greatly reduces the burden on the programmer.

A similar problem arises in the implementation of finite element methods on many-core architectures. Using a greater number of smaller kernels increases the total number of threads that can run concurrently and increases performance. However, using more kernels requires more intermediate storage arrays. In this case, the use of memory bandwidth rather than memory space becomes an issue, as the performance of the kernels becomes limited by their ability to transfer data to and from global memory. Out of these possible implementations, a search is required to find the optimum kernel granularity subject to the constraints on the occupancy of individual kernels.

3.4.2 SPIRAL

SPIRAL [PMJ+05] is a platform for generating highly efficient code for *Digital Signal Processing* (DSP) transforms. The motivation for its development is the observation that low-level compilers perform sub-optimally at optimising low-level C code for DSP transforms. Low-level code contains little semantic information about the algorithm it implements, so compilers are forced to make general assumptions about the code, which inhibits optimisation.

In SPIRAL DSP transforms are defined using the *Signal Processing Language* (SPL), whose representation is based on matrix algebra. SPL constructs are manipulated according to a set of rules that allow a complex transform to be rewritten in terms of simpler ones (e.g. the Cooley-Tukey rule), or that perform algebraic rewrites by using matrix identities. These rules may be repeatedly applied to generate a broad class of algorithms that perform the same computation.

The various implementations are converted into an SSA-form [ALSU06] intermediate representation (known as Σ-SPL). Standard optimisations including loop unrolling, intrinsic precomputation, common subexpression elimination, etc., are applied to this representation. These optimised representations are converted to scalar C code or vector code using intrinsics for the target platform. The choice of vector intrinsics is influenced by a description of the available operations on the target hardware.

There is a large variation in the performance of the generated implementations of transforms. In

order to select the best implementations, learning and search techniques are used. The learning framework is used to prune the search space by removing candidate implementations that are expected to perform poorly, based on automatically built performance models of the operations that constitute each implementation. Dynamic programming [Bel03] and evolutionary search [Gol89] are used to find the best candidates in the remaining space of possible implementations.

We remark that there is almost a direct analogy between SPIRAL and the FEniCS tools. In both projects, high-level domain-specific languages are used as a source from which to generate optimised code. It is conjectured that the techniques used in SPIRAL to generate efficient DSP transforms may be translated to generate efficient implementations of finite element codes on many-core platforms. In particular, the idea of using a rulebase to guide transformations of an algorithm is applicable to the finite element method, as demonstrated in Section 3.3.1. The generation of a space of possible implementations of finite element algorithms may also necessitate mechanisms for pruning the search space.

We also note that techniques for searching for the optimal implementation may also be applied at a lower level: for example, there is a space of possible kernel fusions as described in Section 3.2.3, and some mechanism for searching this space will be required. Additionally, characteristics of the target hardware such as the L1 cache size, the number of SMs on the device, and the number of registers, etc., may form part of a performance model that is used to guide this search.

3.5 Stencil Computations

There are various implementations of DSLs for generating and tuning stencil codes on GPUs. These include Mint [UCB11], which has been applied to accelerate a 3D earthquake simulation with minimal changes to the existing application [UZC+12]. The SBLOCK framework [BP10] provides a DSL embedded in Python for defining stencil computations that are translated into low-level GPU kernels. In [ZM12], a stencil computation framework that performs auto-tuning of the alignment of data structures and kernel launch parameters on GPUs is developed. Pochoir

[TCK+11] is a compiler for a domain-specific stencil language embedded in C++, compiling down to a parallel cache-oblivious algorithm in Cilk targeting multi-core CPUs. Performance portability of automatically tuned parallel stencil computations on many-core architectures is demonstrated by [KCO+10], generating code from a sequential stencil expressed in Fortran 95, and the code generation and auto-tuning framework PATUS [CSB11] with stencils specified in a DSL embedded in C. Although all these DSLs successfully generate optimised code, they are limited to structured meshes. A key advantage of the finite element method is its ability to handle complex geometries using unstructured meshes, and for this reason, we neglect to discuss further the tools which can only handle the structured case.

3.6 Conclusion

We have examined finite element implementations for many-core architectures and discussed some of the performance optimisations and strategies for maximising parallelism on these devices.

We have examined code generation frameworks for the finite element method and discussed how they can facilitate the optimisation of codes by automatically generating transformed variants of a solver from high-level source code.

Finally, we have discussed abstractions and frameworks that have been used in other areas of scientific computing. These abstractions all allow developers to be freed from optimisation process to some degree and allow them to focus instead on the application domain.

Chapter 4

Global Assembly Strategies on Multi- and Many-core Architectures

4.1 Introduction

In this chapter we demonstrate that performance portability in the finite element method cannot be achieved through the use of a single source code written in a low-level language. Although platform portability is provided by OpenCL, it does not achieve performance portability. We demonstrate this by exploring different global assembly strategies and data layout transformations. The discussions and experimental results presented in this chapter were published in [MSH+13], which builds upon a previous publication [MHK10], in which a proof-of-concept implementation of global assembly algorithms using written was presented. The contributions of this chapter are:

- A thorough mapping of the implementation space of existing algorithms for global matrix assembly in low-order finite element methods on multi- and many-core architectures. This exploration demonstrates that different choices that are made at a more abstract level than can be represented in a single low-level source code are required when implementing finite element assembly on different multi- and many-core architectures.

- A demonstration of high-performance implementations of a finite element advection-diffusion solver written using CUDA and OpenCL. These implementations give an order of magnitude speedup on an NVidia GTX480 when compared to an optimised baseline implementation running on a 4-core Intel Xeon E5620 Westmere CPU.

The conclusions of the experiments presented in this chapter are relevant to those working with the finite element method, and can be used to influence the development of their codes. This includes those implementing high-order finite element methods [SSD03], spectral/hp-type methods [KS99], and low/high order Discontinuous Galerkin methods [HW07].

4.1.1 Summary of Results

We will see that a data layout that destroys temporal locality but increases coalescing can increase performance on many-core architectures. Additionally, the use of a global assembly algorithm that increases the level of redundant computation but greatly reduces the amount of uncoalesced accesses also increases performance on many-core architectures.

This motivates variation in the code generated by an automated finite element assembly toolchain - in particular, the code must be customised to use a data layout that is most suited to the target architecture. Additionally, the choice of global assembly algorithm must be made such that the optimal choice for the target hardware is used - this could be achieved by swapping the implementation of a matrix assembly and multiplication for alternatives. This choice should lie beneath the abstraction presented by the code generator.

We continue this chapter by describing the global assembly phase in detail and considering its implementation on many-core architectures. Subsequently we describe an experimental evaluation of the performance of different algorithms and data formats on multi- and many-core architectures, and finally draw conclusions.

4.2 The Global Assembly Phase

The global assembly phase is a performance bottleneck in many finite element applications. Investigations of global assembly algorithms [CSKK11b, CSKK11a, VSK10] show that depending on the equation being solved, the domain over which it is solved, and the choice of basis functions, the point at which it is profitable to switch algorithms for global assembly varies depending on the problem. We shall examine the implementation of the global assembly phase as it is a portion of the computation that makes up a substantial proportion of the execution time of finite element solvers, is difficult to optimise on many-core architectures, and is used in a wide variety of finite element implementations.

Recall from Section 2.2 that in the global assembly phase, the local matrices, $\mathbf{M}^e$, and vectors, $\mathbf{b}^e$, are used to form a global matrix, $\mathbf{M}$, and global vector, $\mathbf{b}$, respectively. This is achieved by performing the following computations:

$$\mathbf{M} = \mathcal{A}^T \mathbf{M}^E \mathcal{A}, \tag{4.1}$$

$$\mathbf{b} = \mathcal{A}^T \mathbf{b}^E, \tag{4.2}$$

where $\mathcal{A}$ is a matrix mapping the local node numbers of each element to the global node numbers, $\mathbf{M}^E$ is a block-diagonal matrix whose e-th block is $\mathbf{M}^e$, and $\mathbf{b}^E$ is a vector of stacked $\mathbf{b}^e$. The different global assembly algorithms all provide different mechanisms for implementing these computations. Consider a two-element, three-node decomposition of a one-dimensional domain (see Figure 4.1). In this example, the matrices and vector are:

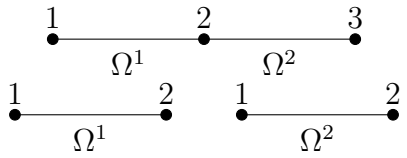


Figure 4.1: *Top:* A 1D domain decomposed into two elements (Ω^i). *Bottom:* Local node numbering of individual elements.

$$\mathcal{A} = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & 1 & & \\ & & & 1 \end{bmatrix} \qquad \mathbf{b}^E = \begin{bmatrix} b_1^1 \\ b_2^1 \\ b_1^2 \\ b_2^2 \end{bmatrix} \tag{4.3}$$

$$\mathbf{M}^E = \begin{bmatrix} m_{11}^1 & m_{12}^1 & & \\ m_{21}^1 & m_{22}^1 & & \\ & & m_{11}^2 & m_{12}^2 \\ & & m_{21}^2 & m_{22}^2 \end{bmatrix} \tag{4.4}$$

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

It is usually inefficient to compute the matrix multiplications described in Equations 4.1 and 4.2 due to the sparsity of $\mathcal{A}$. The *Addto* algorithm is usually more efficient. To describe this algorithm, we first define an array, $\texttt{map}[e][i]$, that maps the local node i of the element e to a global node number. In our example, the array is defined as:

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

Figures 4.2 and 4.3 describe the *Addto* algorithm for computing $\mathbf{M}$ and $\mathbf{b}$. Intuitively, terms of the local matrix (or vector) of each element are summed into particular terms in the global

```
M = 0
for each Element e do
  for i ← 1 to N do
    for j ← 1 to N do
      M[map[e][i], map[e][j]]+=M^e[i, j]
    end for
  end for
end for
```

Figure 4.2: Addto for global matrix assembly.

```
b = 0
for each Element e do
  for i ← 1 to N do
    b[map[e][i]]+=b^e[i]
  end for
end for
```

Figure 4.3: Addto for global vector assembly.

matrix (or vector) depending on the node numbers of the element. We note that the loop nest that implements the algorithm relates to the algebra from which it is derived - in particular it performs the gather described by the structure of $\mathcal{A}$ in Equation 4.1.

4.2.1 Implementation on Many-core Architectures

The Addto algorithms for global assembly are massively data-parallel, as iterations of all the loops can be executed independently of others. Although this appears to make them ideal for implementing on GPU architectures, there are two issues. First, data races occur if threads concurrently update the same term of the global matrix. Atomic operations or colouring must be used to ensure correctness. Second, $\mathbf{M}$ is often stored using a format such as *compressed sparse row* (CSR). Finding the location in memory of a particular term requires a bisection search of the sparsity structure of the matrix, leading to uncoalesced accesses and control flow divergence within warps (see Section 2.6).

To avoid these issues, we can derive an alternative algorithm, referred to as the *Local Matrix Approach* (LMA) [CSKK11b], noting that the only use of $\mathbf{M}$ is for computation of the product $\mathbf{Mv}$ in the solution phase. We omit the global assembly of $\mathbf{M}$ (Equation 4.1) altogether, and

when computation of $\mathbf{y} = \mathbf{M}\mathbf{v}$ is required, the following computation is performed:

$$\mathbf{y} = \left(\mathcal{A}^T \left(\mathbf{M}^E \left(\mathcal{A}\mathbf{v}\right)\right)\right). \tag{4.5}$$

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

We note that using the Local Matrix Approach instead of the Addto algorithm results in an increase in computation and memory bandwidth usage in the solver phase proportional to the average number of elements that share a single node (the *variance*) of the mesh. However, its implementation avoids the use of atomic operations and bisection searches in the global assembly phase. In [CSKK11b, CSKK11a, VSK10], the optimal choice of algorithm in CPU implementations depends on the problem parameters. We demonstrate in Section 4.3 that the optimal choice of algorithm also depends on the target hardware.

4.2.2 Colouring

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

Performing the colouring reduces the total available parallelism, as only updates of the same colour can execute in parallel. However, in the implementation of the Addto algorithm, the maximum number of colours is equal to the maximum variance of the mesh. Typically, up to 10 colours are required with a 2D mesh, and up to 30 with a 3D mesh. As there will typically be hundreds of thousands of rows in the global matrix, there is still a large amount of parallelism available within each colour. It has been shown in experiments described in [CLD11] that

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

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

$$\mathcal{A} = \begin{bmatrix} 1_0 & & \\ & 1_0 & \\ & 1_1 & \\ & & 1_0 \end{bmatrix} \tag{4.6}$$

$$Set_0 = \{m^1_{1,*}, m^1_{2,*}, m^2_{2,*}\} \tag{4.7}$$

$$Set_1 = \{m^2_{1,*}\} \tag{4.8}$$

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

4.2.3 Data formats

In general, data structures must be carefully chosen to achieve optimal performance (e.g. for cache-optimality), and the optimal choice of data structure depends on characteristics of the target architecture. In order to examine the structures that can be used when implementing

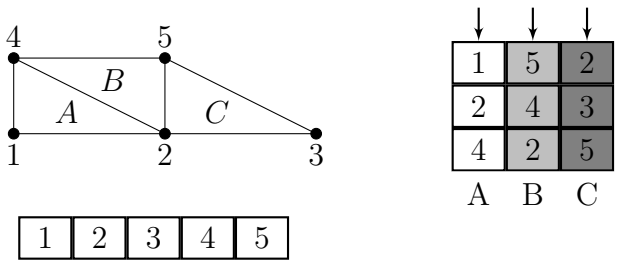


Figure 4.4: *Top:* A 2D domain decomposed into three elements. *Bottom:* Node data layout in CPU implementation. *Right:* Node data layout in GPU implementation. Threads accessing data in different elements (arrowed) achieve coalescing.

the finite element method, we consider a three element domain (see Figure 4.4).

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

This data format is inefficient for many-core implementations, where coalesced accesses must be used to maximise memory performance. It is difficult to achieve coalesced access because the nodal data structure is accessed in a somewhat random fashion. We propose that it can be more efficient to store nodal data on a per-element basis in many-core implementations, interleaving the nodal data for each node of each element. This leads to some redundancy in the storage of nodal data, again proportional to the average variance of nodes; however, it allows coalesced accesses when there is a one-to-one mapping between threads and elements. For the simulations we describe using triangular elements (see Section 4.3), the level of redundancy is a factor of approximately 6. For tetrahedral meshes, the redundancy will be a factor of approximately 20-30, and an alternative data layout may be more efficient in this case.

4.2.4 Implementation of LMA

The Local Matrix Approach is implemented by considering the computation in Equation 4.5 in three stages:

$$\underbrace{\mathbf{t} = \mathcal{A}\mathbf{v}}_{\text{Stage 1}}, \qquad \underbrace{\mathbf{t}' = \mathbf{M}^E\mathbf{t}}_{\text{Stage 2}}, \qquad \underbrace{\mathbf{y} = \mathcal{A}^T\mathbf{t}'}_{\text{Stage 3}}. \tag{4.9}$$

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

4.3 Performance Evaluation

We evaluate the performance of the Addto algorithm and the Local Matrix Approach using an implementation of a test problem that solves the advection-diffusion equation under the assumptions of a divergence-free velocity field, zero source term and spatially-constant isotropic diffusivity (discussed in Section 2.2.5):

$$\frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T = D\nabla^2 T, \tag{4.10}$$

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

$$m(v,T) = \int_\Omega vT \,\mathrm{d}X, \tag{4.11}$$

$$d(v,T) = \Delta t \int_\Omega D\nabla v \cdot \nabla T \,\mathrm{d}X, \tag{4.12}$$

$$a(v,T) = \Delta t \int_\Omega \nabla v \cdot \mathbf{u}T - vT\nabla \cdot \mathbf{u} \,\mathrm{d}X, \tag{4.13}$$

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

$$m(v,T_1) = a(v,T^n), \tag{4.14}$$

$$m(v,T_2) = a(v,T^n + \tfrac{1}{2}T_1), \tag{4.15}$$

$$m(v,T_3) = a(v,T^n + \tfrac{1}{2}T_2), \tag{4.16}$$

$$m(v,T_4) = a(v,T^n + T_3), \tag{4.17}$$

$$T^a = T^n + \tfrac{1}{6}T_1 + \tfrac{1}{3}T_2 + \tfrac{1}{3}T_3 + \tfrac{1}{6}T_4, \tag{4.18}$$

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

$$m(v,T^{n+1}) + \tfrac{1}{2}d(v,T^{n+1}) = m(v,T^a) - \tfrac{1}{2}d(v,T^a), \tag{4.19}$$

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

$[-1.2, 1.2] \times [-1.2, 1.2]$. The initial tracer concentration is described by:

$$T = \begin{cases} 1 & \text{if } r < \frac{1}{4} \\ \\ 0 & \text{otherwise,} \end{cases} \tag{4.20}$$

where r is the distance from the point $(-0.5, 0)$. The velocity $V = (1, 0)$ and the diffusion coefficient $D = 0.1$ at all points in the domain.

4.3.1 Experimental setup

Our test hardware consists of three different GPUs. The first, an NVidia GTX280, is an implementation of the Tesla architecture. The second, an NVidia GTX480 is an instance of the Fermi architecture. Third, we use an AMD Radeon 5870, which is an instance of the Evergreen architecture. The CPU used to test the baseline implementations was an Intel Xeon E5620 (based on the Nehalem architecture) with 12GB of RAM.

CUDA and OpenCL implementations of the solver that implement both the Addto algorithm and the Local Matrix Approach have been produced for execution on the GPUs. The baseline version is implemented within Fluidity [App10], a finite element computational fluid dynamics code that has been chosen because it is a mature and optimised CPU implementation that is in production use for scientific applications, such as the Imperial College Ocean Model (ICOM) [Pig10]. The Local Matrix Approach is not implemented in this version, as it is known to be less efficient than the Addto algorithm on CPUs for the low-order basis functions used in these experiments [VSK10]. Execution is parallelised in Fluidity using domain decomposition, and different processes communicate using MPI. Node data structures are implemented using the element-wise storage layout in the CUDA implementation, and the node-wise layout is used in the CPU implementation.

The Intel v10.1 compilers with the `-O3` flag were used to compile the CPU code (v11.0 onwards could not compile Fluidity at the time due to compiler bugs), and the CUDA SDK 3.1 is used

for the CUDA and OpenCL code running on NVidia GPUs. The AMD Stream SDK version 2.2 was used to compile the OpenCL for the AMD GPU, and to compile the OpenCL code to run on the CPU. The CUDA and OpenCL implementations use a *Conjugate Gradient* (CG) solver described in [MK09]; the baseline version makes use of the PETSc version 2.3.3 [BBG+09] CG solver. The simulation is run for 200 timesteps, with all computations using double precision arithmetic. Gmsh [GR09] was used to generate meshes varying in size between 58485 and 509260 elements. Using these mesh sizes allows the experiments to execute inside 1GB of RAM, which is the maximum available on the NVidia GTX280 and AMD 5870. Each simulation was run five times, and averages are reported.

4.4 Results

In this section we discuss the performance of both algorithms on all of the target platforms. In particular, we note that the Local Matrix Approach is faster than the Addto algorithm across all the GPUs, whereas the Addto algorithm is faster on the CPU.

First, we present the results from the OpenCL implementation on the AMD Radeon 5870 in Figure 4.5. The largest mesh that could be used on this hardware consisted of 367321 elements, since the current driver implementation only allows 768MB of memory to be allocated. Only the OpenCL implementation could be run on this hardware, as CUDA is not supported.

Next, we examine the performance of the CUDA and OpenCL implementations on the NVidia GTX280 in Figure 4.6. It can be seen that for some mesh sizes the OpenCL implementation outperforms the CUDA implementation and vice-versa for others. However, the difference in execution times on each mesh is negligible. Although both implementations of the Addto algorithm using colouring perform worse than the respective LMA version, the OpenCL implementation of this algorithm is noticeably slower than the CUDA implementation. We hypothesised in [MHK10] that the LMA would outperform the Addto algorithm with colouring; these results validate this hypothesis. We note that using atomic operations instead of colouring leads to a further reduction in performance.

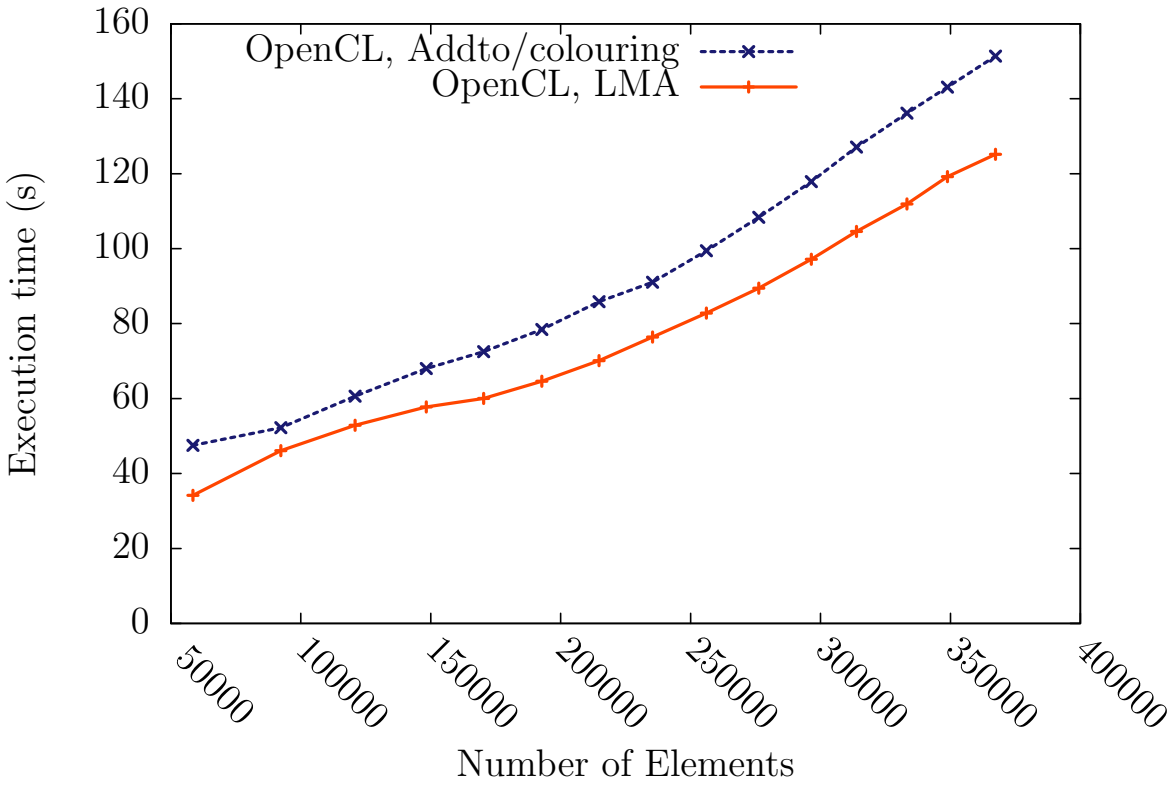


Figure 4.5: Execution times for simulations on the AMD Radeon 5870. Note that the Local Matrix Approach is more efficient on this hardware.

Considering the NVidia GTX480 in Figure 4.7, we see that the performance of atomic operations is much increased on this architecture, and the execution times of the Addto algorithm with atomics and colouring are very similar. However, the LMA implementations are still more efficient than the Addto implementations. In the LMA implementations, there appears to be a constant overhead of using OpenCL compared to using CUDA. The OpenCL implementation of the Addto algorithm using colouring again is far slower than the OpenCL LMA implementation. The gap in performance between the Addto and LMA in CUDA is much smaller in this architecture - it was suspected that the L1 cache that is present in the NVidia GTX480 may have reduced the overhead of the irregular accesses required by the Addto kernel. However, use of the NVidia Compute Profiler [NVi12b] showed that the L1 cache hit rate was less than 5% for this kernel.

Next, we consider the execution time of the CPU baseline version on the Intel Xeon E5620 in Figure 4.8. Because of the platform-portability of OpenCL, we were also able to compile and run the OpenCL code intended for the GPU on this platform. We see that the Addto

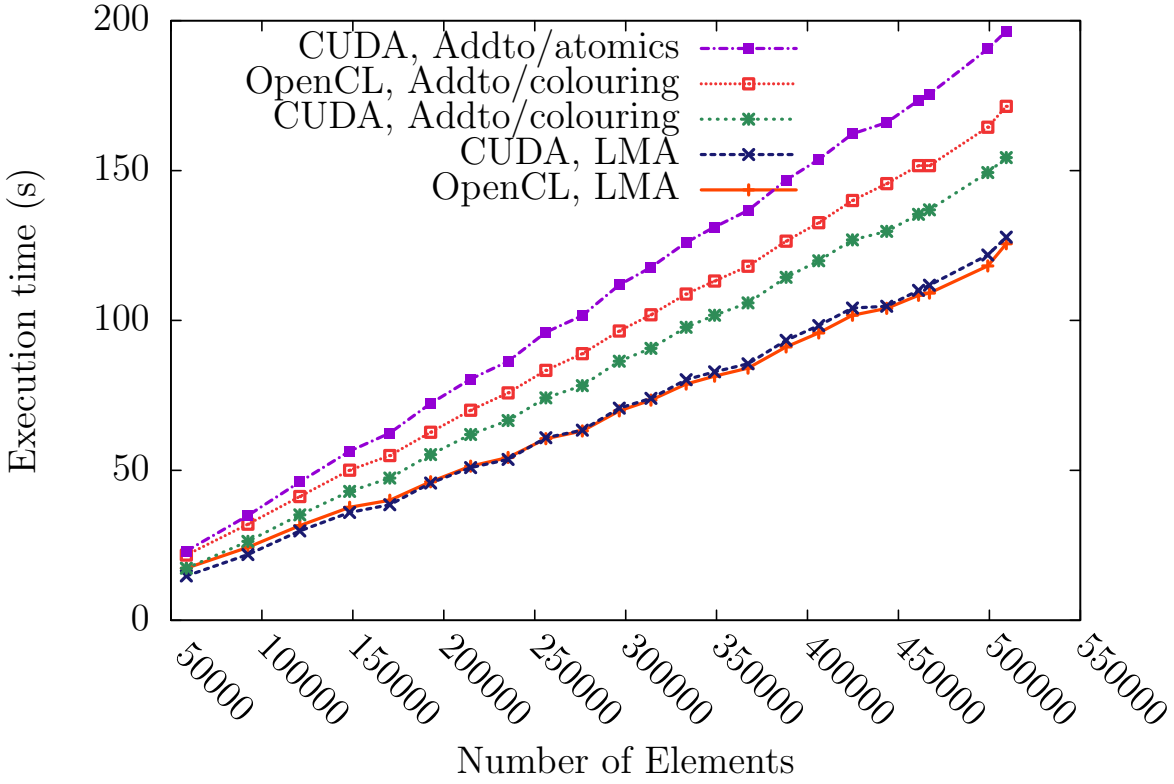


Figure 4.6: Execution times for simulations on the Nvidia GTX280. The CUDA and OpenCL implementations of the Local Matrix Approach are the fastest, and achieve comparable performance.

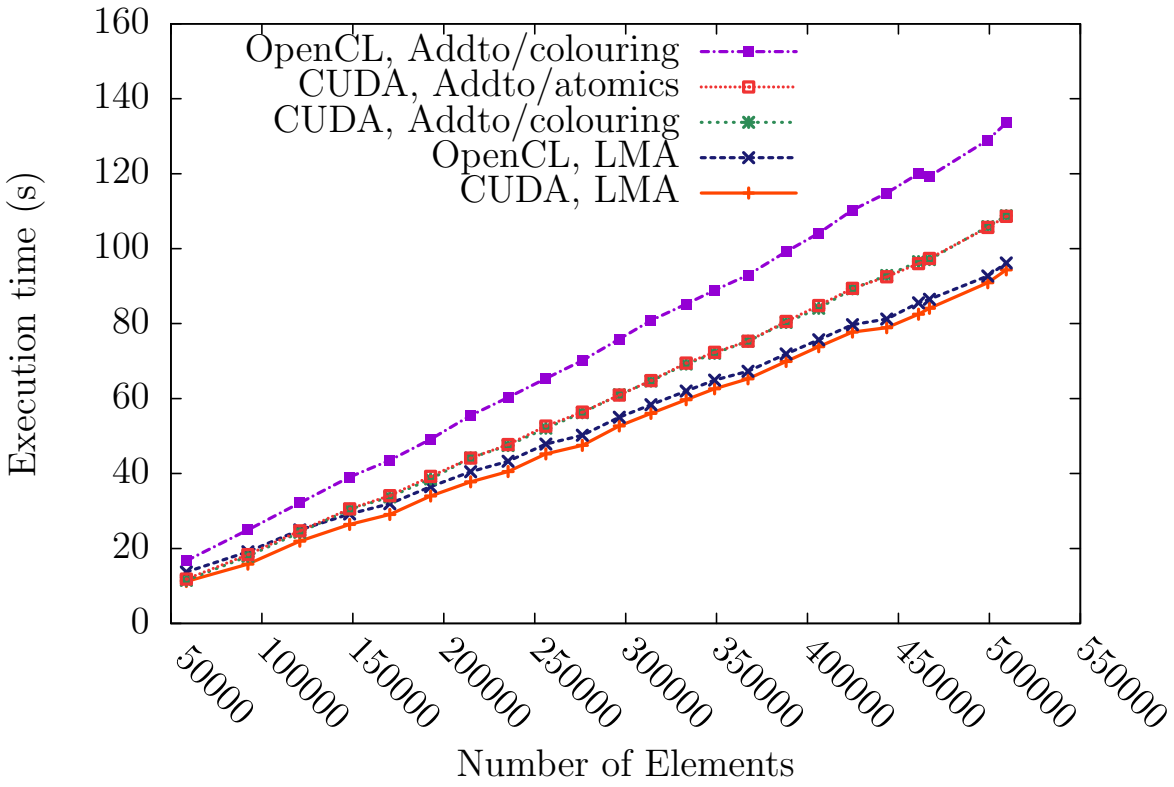


Figure 4.7: Execution times for the simulations on the Nvidia GTX480. Atomic operations appear far more efficient on this hardware than on the GTX280.

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

Figure 4.9 shows the relative speed of the fastest implementation on each architecture. Normalised throughput is computed by dividing the number of elements by the execution time. Note that this may not be taken directly as a measure of elements assembled per second, since the work required within the solve phase varies from mesh to mesh. We observe that the NVidia GTX480 provides a speedup of approximately an order of magnitude over the 4-core Intel Xeon E5620. There appears to be a large start-up cost when using the AMD 5870, which is not amortised until the mesh is on the order of 200000 elements. Additionally we note that its performance is approximately 50% of that of the NVidia 480GTX. This is a surprising result given that the two architectures have similar peak memory bandwidth and double-precision arithmetic throughput [AMD10, NVi10b].

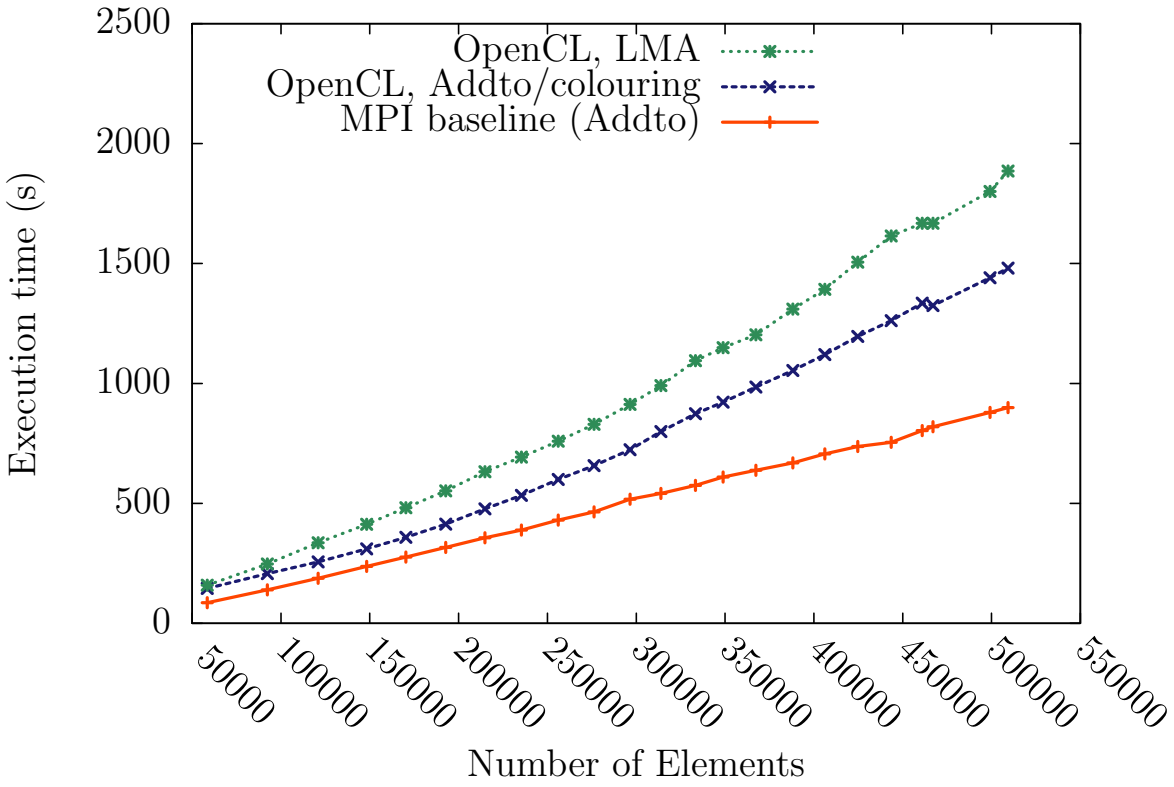


Figure 4.8: Execution times of the simulations on the 4-core Intel Xeon E5620. We see that the Addto algorithm is most efficient on this hardware. The the performance of the OpenCL code is less than the baseline, which is expected to be due to its use of expanded data layouts.

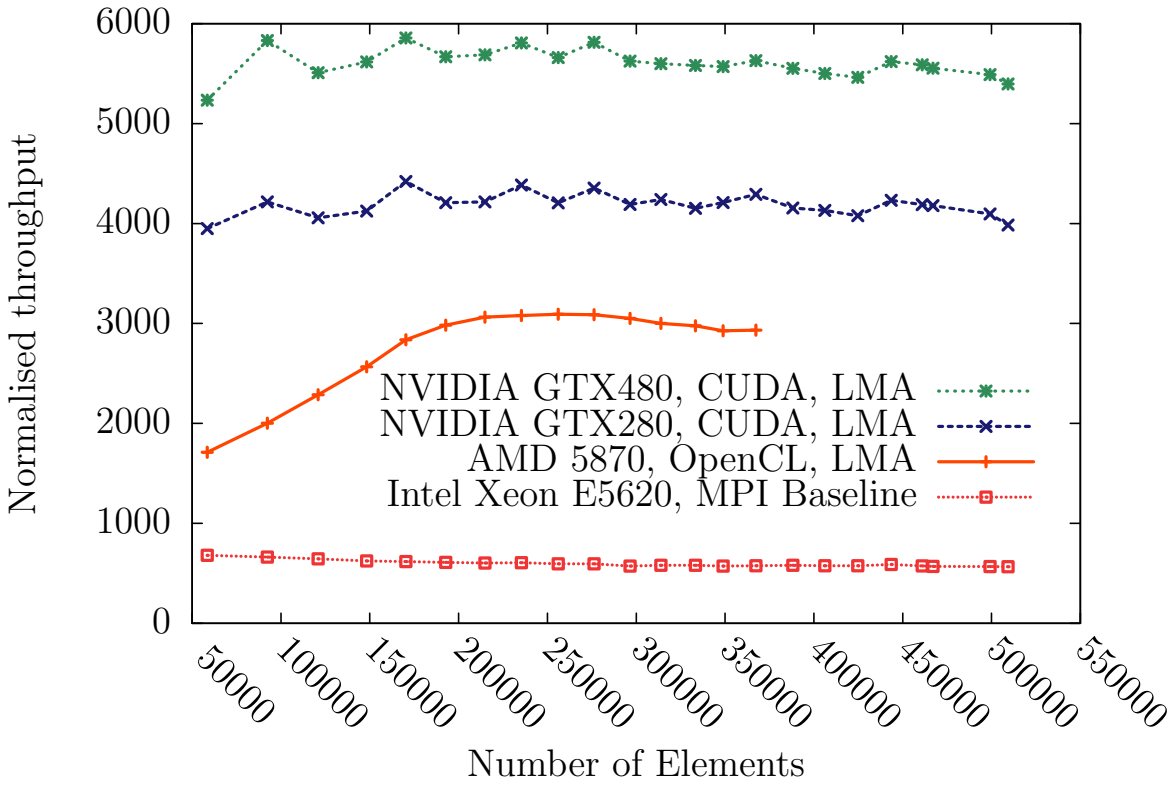


Figure 4.9: Normalised throughput of the best implementations on each architecture (higher is better). We see a performance gain of an order of magnitude between the NVidia GTX480 and CPU baseline implementation.

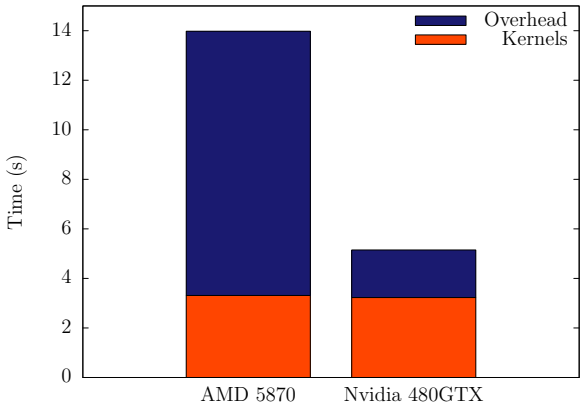


Figure 4.10: Kernel execution times and driver overheads. Kernel execution times are similar, but the driver overhead of the AMD card significantly lowers performance.

In order to investigate the performance on the AMD Radeon 5870, we recorded the start and end execution times of each kernel by using OpenCL events and obtaining the profiling information for the event associated with each kernel invocation. Since recording the profiling information occupies some memory on the device, the largest mesh that this could be performed for was one with 148290 elements, and the simulation could only be run for 50 timesteps. This profiling information showed that the execution time of the kernels was relatively low compared to the entire execution time of the simulation. The sum of the kernel execution times obtained is compared with the total execution time as recorded by timing routines in the host code (which includes driver overhead) in Figure 4.10.

We see that the total execution time for all the kernels are very similar on both architectures. However, the total execution time on the AMD GPU is significantly higher, indicating that the driver overhead of kernel launching is high. We conclude that the implementation of the AMD hardware is competitive with the NVidia hardware. However, the software implementation in its present form is less competitive.

4.4.1 Summary of Results

We conclude from the results from all architectures that the LMA is the fastest algorithm on many-core architectures. This is due to the increased coalescing and reduced control flow divergence afforded by this algorithm. On the CPU architectures, the Addto algorithm is

the fastest approach, as a result of the larger cache, and overall lower memory bandwidth. The poor performance of the LMA in our OpenCL implementation compared to the Addto algorithm on the CPU agrees with the results of previous studies using other languages for their implementation [CSKK11b, CSKK11a, VSK10]. It is also apparent that the OpenCL implementation is performance portable across similar architectures, but not across the gap between multi-core and many-core architectures.

4.5 Conclusions

We have investigated the implementation space for parallel global assembly in the finite element method using a variety of many-core platforms and a multi-core architecture. These simulations were performed using an implementation of a finite element advection-diffusion solver.

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

We have demonstrated that although the OpenCL implementation is portable across CPU and GPU platforms, optimal performance cannot be achieved on all architectures without modifying the source code. This motivates the automation of code generation, since a single low-level implementation is not sufficient to achieve performance portability. In the following chapter we describe a prototype code-generation tool that transforms UFL sources into CUDA implementations for execution on NVidia's many-core architectures.

Chapter 5

The Manycore Form Compiler

5.1 Introduction

In this chapter we describe the *Manycore Form Compiler* (MCFC), which was developed as a step on the way towards building a form compiler that generates OP2 code. The OP2 abstraction was chosen because it provides mechanisms for handling the following concerns:

1. The necessity to generate code using different data layouts and algorithms depending on the target architecture, as demonstrated in the previous chapter.

2. Abstraction of the target hardware from the generated code using an intermediate representation. This separates the concerns of translating high-level UFL forms into imperative code from the efficient mapping of this code onto the target hardware.

3. Integration of the generated code into a large Fortran 90 codebase, namely Fluidity.

OP2 is well-suited to handling the first two concerns since it abstracts the parallel execution and data formats from the code written for its API. The third concern is also handled within the scope of OP2, since it has been designed with integration into the HYDRA CFD code, which is written in Fortran, as a requirement.

Although OP2 is well-suited for parallel computations on unstructured meshes, it lacks some of the abstractions necessary for its use in a finite element toolchain, particularly for matrix assembly and linear algebra. In order to understand how the OP2 API should be modified to address these shortcomings, the development of MCFC focused on the direct generation of CUDA code.

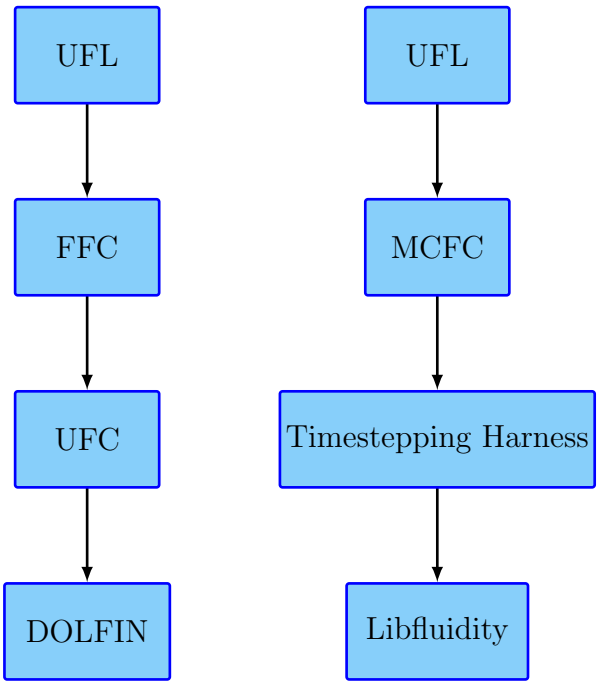
Ultimately we learn from the development of MCFC that the static translation approach, which is used in OP2, is not well-suited for integrating methods specified in UFL into a pre-existing codebase. Since UFL is embedded in a dynamic language, robust translation is difficult since it is not possible to easily anticipate how the execution of the simulation code will proceed at runtime - this is explained in more detail in Section 5.7.1. Eventually the development of MCFC was discontinued in favour of the Firedrake toolchain, which we describe in Chapter 7.

We continue this chapter by outlining the architecture of MCFC, and will examine each of its phases in detail. In particular, we present algorithms for minimising the code size of local assembly kernels. These algorithms provide a means to generate a local assembly loop nest and place individual small expressions within particular loops. This differs from the strategy currently used in FFC, which generates a single very large expression for computing each entry in a local tensor.

We evaluate the code generated with MCFC by comparing it with code generated by DOLFIN. We conclude this chapter with a discussion of recommendations for modifying OP2 so that it is more suited as a parallel execution layer in a finite element form compilation toolchain.

5.2 MCFC Architecture

The design of the MCFC is influenced by the design of FFC and the other components of the FEniCS toolchain. Figure 5.1 gives an overview for comparison of the two toolchains. In FEniCS, FFC generates code from UFL sources that conforms to the UFC interface. DOLFIN links in the UFC-conformant code and makes calls to the functions defined within.



(a) FEniCS/DOLFIN (b) MCFC/Fluidity

Figure 5.1: Form compilation toolchains.

Although it would be possible to make modifications to DOLFIN to fit it into the MCFC toolchain, Fluidity is targeted as part of the goal to apply the work of this thesis to its long-term development. The Libfluidity component at the bottom of the MCFC stack is a library that contains all of Fluidity's finite element implementation code as well as supporting routines. Use of this library serves both as a prototype for Fluidity integration, and allows simulation setup and other non-finite element operations to be delegated to the library.

The UFC intermediate representation is not used in the MCFC toolchain. The UFC specification places constraints on the generated code that are not amenable to efficient execution on many-core architectures, partly because of its specification in C++. The Timestepping Harness that takes the place of UFC is a small Fortran program that uses Libfluidity to initialise the state of a finite element simulation from an option file, and then executes a timestepping loop to advance the simulation in time. The timestepping harness acts as a reduced prototype of the entire Fluidity simulation, which allows for experimenting with the integration of code generation without the complexity of dealing with the full mechanisms used in Fluidity.

Figure 5.2 gives an overview of the control flow of the timestepping harness. The harness retains

only the field values from the previous timestep. This is sufficient for implementing single-step time discretisations such as the Euler method, the Theta method and Runge-Kutta methods [HNW93], since its intermediate values are all contained within a single timestep. Multi-step methods such as Adams-Bashforth can also be implemented, but extra fields must be defined to store the field values from timesteps earlier than the most recent one. When the simulation is completed, the state is written out to disk using Libfluidity. Calls to generated functions that implement the finite element discretisation are embedded in the timestepping harness.

The generated code provides an interface for initialisation. The harness passes the state of the simulation that it has constructed to the initialisation function, which allocates space on the GPU and copies data structures into global memory.

5.3 User Interface

Although the MCFC and FEniCS toolchains comprise different components, source-compatibility with DOLFIN is retained where possible. The portions of code describing the finite element forms in UFL remain unchanged when translating DOLFIN code for use in the MCFC toolchain. Only the code which specifies the source of data used in the computations needs to be changed.

The UFL interface to Fluidity is designed to resemble the Python state interface that it already provides. Fields are accessed using the `state` object. Objects retrieved from the Fluidity state can be used directly in UFL forms, as if they were instances of the UFL `Coefficient` class. This is similar to the way that DOLFIN allows a user to use `Function` objects in a form.

Figure 5.3 gives an example of a solver for an advection equation implemented in Fluidity using MCFC. In this example, the Fluidity options file is assumed to contain at least two fields, `Tracer` and `Velocity`.

Test and trial functions are defined on the same space as the `Tracer` field. It is not normally permitted in UFL to construct these functions directly from a `Coefficient`. To enable this behaviour, the MCFC API wraps the Test and Trial function constructors with another function

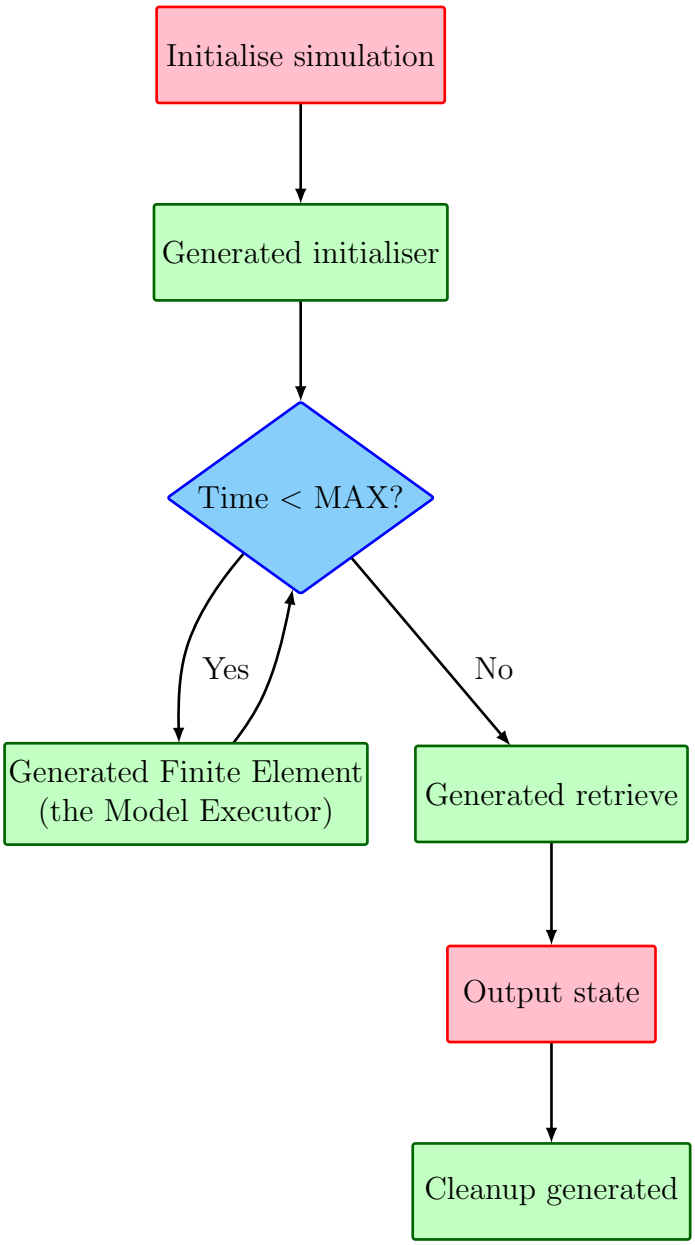


Figure 5.2: Control flow of the timestepping harness. Functions performed by calling Libfluidity are filled in pink, and those performed by generated code are filled in green.

```
t=state.scalar_fields["Tracer"]
u=state.vector_fields["Velocity"]

p=TrialFunction(t)
q=TestFunction(t)

Mass=q*p*dx
rhs = (q*t+dt*dot(grad(q),u)*t)*dx
tnew=solve(Mass,rhs)
state.scalar_fields["Tracer"]=tnew
```

Figure 5.3: Solving the advection equation for a single timestep in Fluidity using MCFC. Field data is retrieved from state and can be treated as `Coefficient` objects. Solutions are assigned back to state at the end of the timestep.

that extracts the element on which coefficients are defined, then passes it to the UFL constructor functions. This is syntactic sugar which retains compatibility with DOLFIN and simplifies the user's code.

The remaining lines define a finite element discretisation of an advection equation advanced in time using Euler's method. The variable `dt` is always in scope, and contains the current length of the timestep. Since Fluidity uses adaptive timestepping, the value can change between timesteps but this is not a concern for the user.

The `solve` function returns a new field. This field is re-assigned to the `Tracer` field at the end of the timestep. The state of any new fields created as the result of a solve but not stored back into the state are discarded. For example, if a Runge-Kutta scheme is used to advance the solution in time, only the field values at T^{n+1} are retained, and field values at the intermediate steps discarded.

5.4 The MCFC Pipeline

The MCFC pipeline statically compiles a finite element discretisation and its coupling to another numerical method by parsing and analysing a Python script that expresses the computations it performs at each timestep. The phases are:

Code extraction. The Fluidity options file is loaded and parsed in order to extract the UFL code strings. This is achieved using SPUD [HFG+09], a library for manipulating Fluidity option files. SPUD populates a dictionary at runtime that can be queried to discover which options were set and their values.

Execution. The preprocessed code is executed within a namespace, referred to as the *simulation namespace*, containing the Fluidity state. After execution, the namespace holds objects that represent the computations used in the simulation.

Form preprocessing. The form preprocessing functionality of the UFL library is invoked on forms in the simulation namespace.

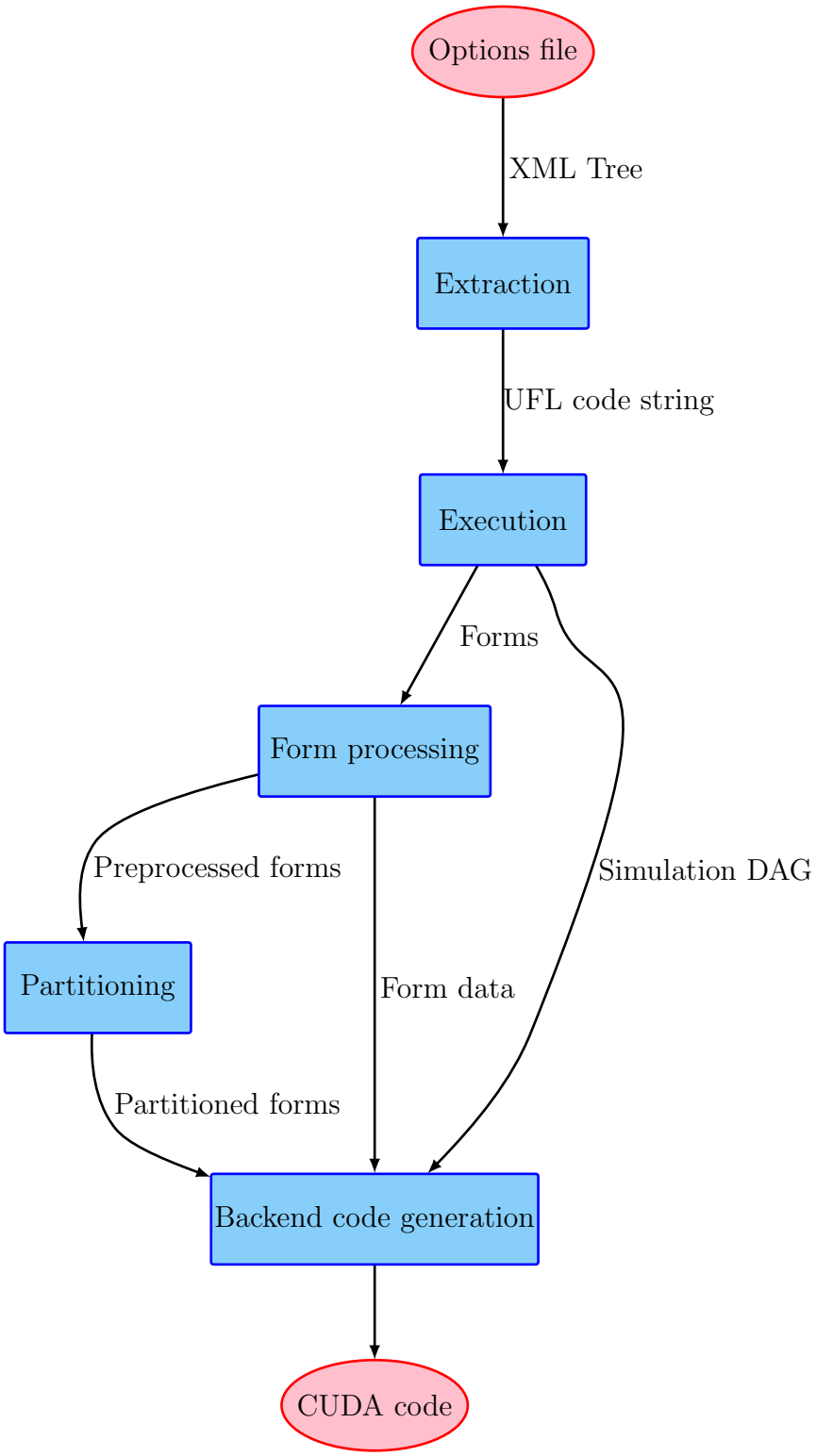


Figure 5.4: Data flow between MCFC phases.

Partitioning. Loop induction variables that are used by nodes in the expressions that make up forms are determined in this phase. Expressions are partitioned into subexpressions that share a set of these induction variables. This information is used when local assembly loop nests are generated.

Code generation. Partitioned forms are translated into kernel code. Initialisation code that performs data transfers and simulation setup is generated. Finite element assembler code, which invokes the generated kernels in order to advance the simulation in time, is also generated.

An overview of data flow between the phases is given in Figure 5.4.

5.4.1 Execution

In order to capture all the information that is required to generate CUDA code implementing a method specified in the options file, the following information needs to be captured during its execution:

- Which fields are extracted from the `state` object.
- Which of these fields are used in each form, and which term they are in the form.
- The forms that are used as input to each `solve` function, which fields are produced by it as output.
- Which fields are returned back into the `state` object.

In order to capture the fields that are extracted from and returned to the `state` object, the implementation of the state field dictionaries (`scalar_fields`, `vector_fields` and `tensor_fields`) make a record of each item that is retrieved from them. The objects that they return implement a UFL `Coefficient`, which has a unique ID. This allows the construction of a mapping between field names and `Coefficient` IDs. This provides a mechanism for determining which

Fluidity fields are used in the variational forms. Similarly, when a field is assigned back to state dictionaries, the ID of the returned field and the name of the field it is returned to are recorded.

Many implementations of finite element methods will involve the assignment of a field created by a `solve` to a field in the Fluidity state. In order to keep track of this, the solve function makes a record of the variational forms that were provided to it as arguments. It also constructs a new `Coefficient` object which it records the ID of, and returns.

Combining all the information that is recorded during execution enables the construction of a *Directed Acyclic Graph* (DAG) that represents the computation performed for one timestep in its entirety. Starting at the end of the computation with a Fluidity field that has a `Coefficient` assigned to it, the ID is retrieved. This `Coefficient` ID can be used to locate the invocation of the `solve` function that generated it. Next, the variational forms passed to the `solve` function can then be retrieved. The IDs of `Coefficient` objects used in the variational forms are also retrieved and this information is used to trace back where they were created - either from an earlier solve function, or from being extracted from the `state` during initialisation. The continued process of tracing back the `Coefficient` objects to their sources eventually terminates at the Fluidity fields from which values were retrieved. When the process is complete, a complete graph of data flow between the computations is obtained. This graph is referred to as the *Simulation DAG*.

In order to implement the capture of all the information required to build the Simulation DAG, the simulation namespace is initialised with the following:

- The UFL public interface, i.e. those objects provided by executing `from ufl import *`.
- MCFC wrappers that override UFL Test and Trial function constructors.
- The `state` object, which records accesses to all fields.
- The `solve` function.

After the execution of the simulation, the namespace contains these objects, and all objects created during the execution of the input code. MCFC can then reconstruct the Simulation DAG by iterating over all the objects in the namespace and inspecting them. This provides a straightforward mechanism for constructing the DAG, but has two limitations. Firstly, only the last assignment to a name in the namespace is recorded, so the input code must either be in single-assignment form, or the user must take care to only overwrite unwanted intermediate values. Secondly, although the code executes within an embedded Python interpreter, the execution of the code cannot use all Python semantics - any control flow that depends on runtime values is prohibited. These limitations simplify the process of translating the input code, which is written in a dynamic language, into a form that can be statically translated, but sacrifices the expressiveness of the source language.

5.4.2 Form Processing

In this stage the UFL form preprocessing function is invoked on each form. This converts forms from a syntactic representation of the input into a set of tensor contractions. The tensor contraction representation of a variational form is more easily converted to imperative code as it has a straightforward correspondence to a loop nest containing expressions for computing the terms in the local matrix.

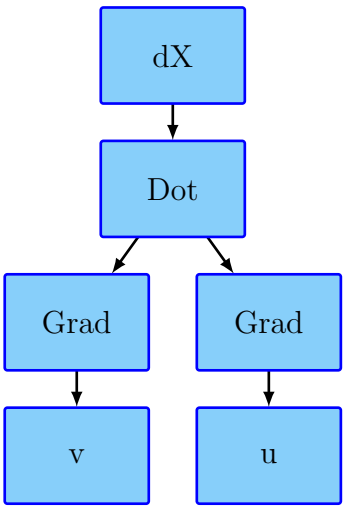


Figure 5.5: Syntactic UFL representation of the Laplacian form.

We give a brief example of the form processing step in order to illustrate the transformation it

performs. For a more complete treatment of the algorithms used in the UFL library, we refer the reader to [ALØ+13].

Figure 5.5 illustrates a syntactic representation of the Laplacian form, $\int_\Omega \nabla v \cdot \nabla u \, \mathrm{d}X$, which is written in UFL as `dot(grad(v),grad(u))*dx`. The DAG shown in the figure is a syntactic representation that is created by the execution of this expression. Translation of this representation of the form to code that implements the local assembly is non-trivial, since the semantics of operators, in this case the dot product, must be implemented by the code generator.

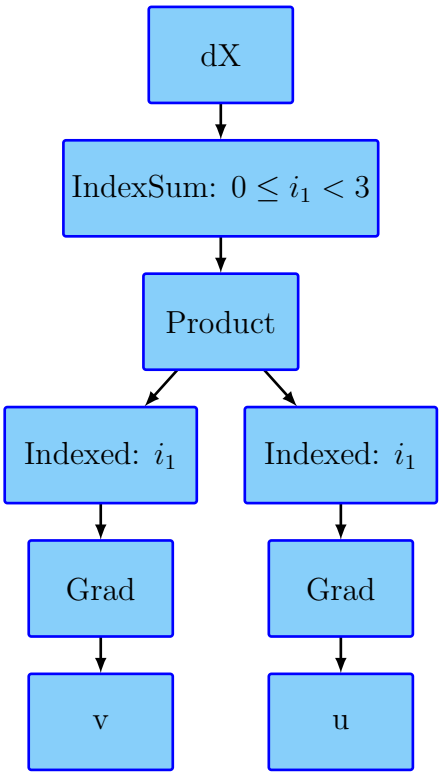


Figure 5.6: Tensor contraction representation of the Laplacian form. Not the `Dot` node has been replaced by a sum of terms over an index (the `IndexSum` node, where the terms are the product of indexed terms.

Figure 5.6 shows the representation of the form after preprocessing. Note that the `Dot` node has been replaced by an `IndexSum` node over a `Product`. The `IndexSum` represents a loop in the local assembly loop nest, with the induction variable i_1. Assuming three-dimensional space, the loop has three iterations. This variable is used to index both of the `Grad` nodes. Since indexing the gradient (a vector) results in a scalar, the product node simply represents the product of two scalars.

This representation is more straightforward to transform into low-level code, since a visitor can walk the tree, firstly to generate the loop nest, and secondly to generate the expression inside the loop nest. The code generation for more complicated expressions uses this process, but there will be multiple `IndexSum` and `Indexed` nodes with more induction variables.

5.4.3 Expression Partitioning

The size of code resulting from the translation of complicated expressions is an issue in FFC. Although the UFL examples that we have discussed in this thesis are relatively simple, it is possible to create much more complicated forms when using mixed finite element spaces or automatic differentiation. One example exhibiting this problem is an implementation of a Fung-type hyperelastic material simulation, where the generated code contains an expression for evaluating local tensor entries that is larger than 45MB [Hak12, Joh11]. In this subsection, we present an expression partitioning algorithm that reduces local assembly code size by enabling the generation of loop nests containing multiple subexpressions, as opposed to generating a single large expression. This requires that subexpressions are placed inside the correct loop; an algorithm for making the correct choice is discussed in Section 5.5.1.

During the code generation process, FFC unrolls all of the loops that are represented by the `IndexSum` nodes in expressions, which increases the size of generated code. In order to ensure that kernels generated by MCFC are of a manageable size, fully-rolled loop nests are generated instead. As well as reducing the code size, the loop nest structure is easier to analyse and can enable further transformations to be made by the CUDA compiler more easily than if it is presented with a single large expression.

In order to generate the rolled loop nest and the expressions contained within, it is necessary to predetermine the indices upon which each term in an expression depends. This information is used to place the terms into the correct loop in the nest. The Expression Partitioning stage in MCFC constructs partitions of an integrand where the nodes in each partition depend on the same set of induction variables. This makes it straightforward for the code generator at a later stage to generate an expression and place it inside the correct loop.

```
def partition(tree):
    nodetype = type(tree)
    if nodetype in (Product, Sum):
        l = findIndices(tree.children[0])
        r = findIndices(tree.children[1])
        if l == r:
            # This node is the root of a partition
            return SubExpr(tree)
        else:
            # Replace children with partitioned children
            children = []
            for child in tree.children:
                children.append(partition(child))
            return nodetype.new(children)
    else:
        # This node is the root of a partition
        return SubExpr(tree)
```

Figure 5.7: Pseudocode for finding all the indices used in a UFL expression tree.

```
def findIndices(tree):
    indices = set()
    for child in tree.children:
        childindices = findIndices(child)
        indices = indices.union(childindices)
    if type(tree) is IndexSum:
        for i in tree.indices:
            indices.add(i)
    return indices
```

Figure 5.8: Pseudocode for finding all the indices used in a UFL expression tree.

Partitioning is performed by the algorithm given in Figure 5.7. Beginning with the root of the tree, the algorithm checks whether the current node is at the root of a partition. A node is at the root of a partition if it is a `Sum` or `Product` node, whose children both use the same sets of indices in all their nodes. If this is the case, then a `SubExpr` node is inserted in the tree at this point, which marks the root of the partition. If the children use different sets of indices, then the children must be partitioned.

If the partition function reaches a node that is not a `Sum` or `Product` node, then the node and its children form an entire partition. This is because all of the `Sum` and `Product` nodes that could have children with different indices are pushed to the top of the tree during its construction.

Finding the indices used by a tree is accomplished using the algorithm given in Figure 5.8. This algorithm descends through the expression and records the induction variables used by

any `IndexSum` nodes that it encounters. This is sufficient to capture all indices, since new indices are only introduced by `IndexSum` nodes.

Example

In order to exemplify the partitioning algorithm, we will partition a form from the finite element discretisation of the Helmholtz equation. Figure 5.9 shows a simplified representation of the expression tree for this form.

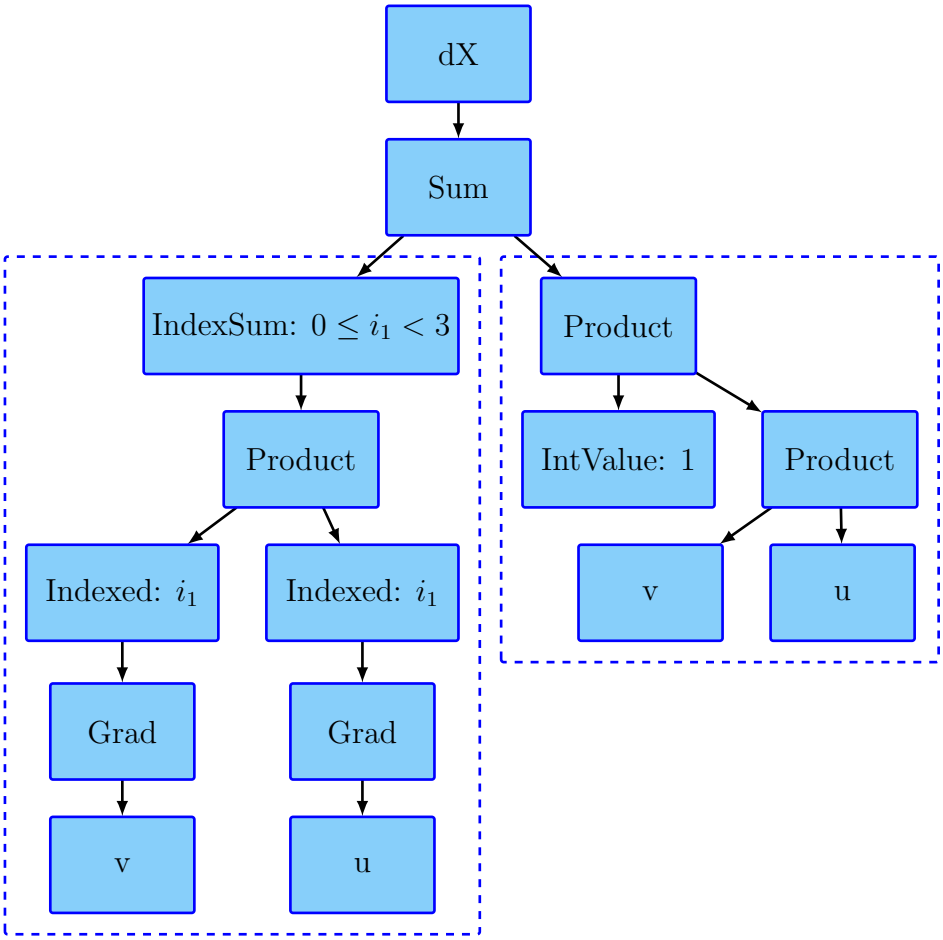


Figure 5.9: Expression tree for the integrand in $\int_\Omega \nabla v \cdot \nabla u + \lambda v u \, \mathrm{d}X$. Nodes that depend on the same set of indices are within the same dotted box.

The partitioning algorithm begins by examining the `Sum` node. The indices used by its children must be examined. The left child node (`IndexSum`) makes use of the index i_1, so its index set is $\{i_1\}$. The other child does not contain any `IndexSum` nodes, so its index set is $\{\}$. Since its children have differing index sets, the sum node cannot be the root of a complete partition

itself. The algorithm must now examine each of the children in turn.

Considering the `IndexSum` node, it is neither a `Product` nor a `Sum` node, so it and its children must all use the same set of indices, and therefore form a partition. The algorithm creates a new `SubExpr` node whose child is the `IndexSum`, which becomes the new child of the `Sum`. The `Sum`'s other child, the `Product`, has two children that both use the same index set, so it is also the root of a partition, and therefore a `SubExpr` node is inserted here. The resulting expression tree is shown in Figure 5.10.

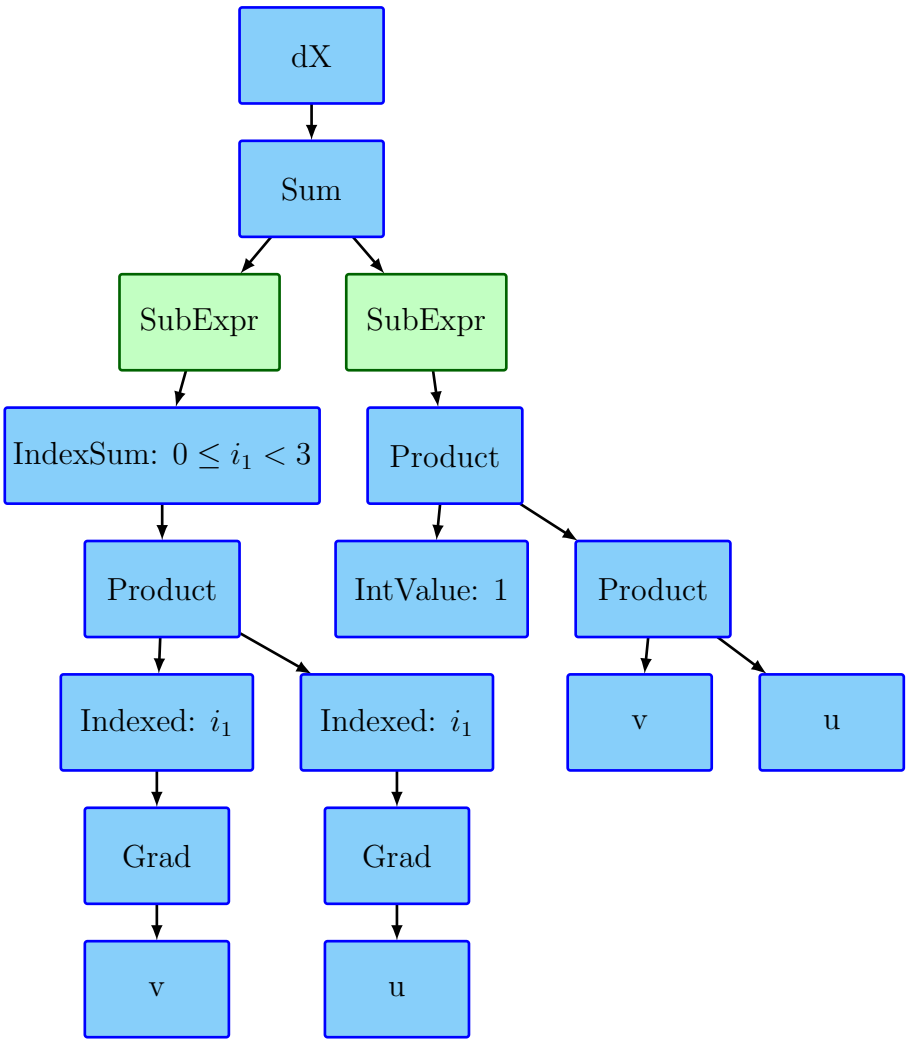


Figure 5.10: Partitioned Intermediate Representation of the term $\int_\Omega \nabla v \cdot \nabla u + \lambda v u \, \mathrm{d}X$.

5.5 Code Generation

MCFC generates the following distinct portions of code that fit into the placeholders in the timestepping harness:

Local assembly kernels. These evaluate the entries of local tensors.

The Model Executor. This contains series of calls to local and global assembly kernels, and the linear solver.

Initialisation code. This transfers the initial state of the simulation to the GPU.

Data transfer code. This is for retrieving the simulation state from the GPU when output is required.

Cleanup code. At the end of the simulation, this code is called in order to free the data structures on the GPU.

The generation of the local assembly kernels is analogous to the generation of the `tabulate_tensor` function by FFC. In addition, FFC also generates code for representing meshes, finite elements, and other entities required in the UFC specification. These portions of code are not required in the MCFC toolchain as they are implemented by Libfluidity and the timestepping harness.

5.5.1 Local Assembly Kernel Generation

The generation of local assembly kernels in MCFC is generally similar to the generation of the `tabulate_tensor` function in FFC. One difference is in the generation of the parameter list. Instead of passing coefficient values through doubly-indirected arrays (as in FFC), an individual pointer to the coefficient data for each field is passed as to the kernel as a parameter. This reduces the number of pointer dereferences required in the kernel.

```
(Loops over local tensor entries)
  <Local Tensor Initialisation>
  (Loop over quadrature points)
    (Loops over reduction indices)
    <Local Tensor Entry Assignment>
```

Figure 5.11: Loop nest structure in local assembly kernels.

The main difference is in the generation of the loop nest that contains the expressions for computing local matrix entries, since loop nests containing multiple subexpressions are generated. In the remainder of this section, we outline an algorithm that generates the loop nests and places subexpressions within, taking as input a partitioned expression. The local tensor assembly loop nest consists of loops over the following indices:

Local Tensor Entries. The number of loops over local tensor entries is determined by the rank of the form. Rank 2 forms have loops over i and j, whereas rank 1 forms have a single loop over i. The extent of these indices is equal to the number of basis functions per element in the test and trial spaces for i and j respectively.

Quadrature points. All integration in MCFC is performed using numerical quadrature. The number of quadrature points is derived from the quadrature degree, which is specified by the user.

Reductions. Each of the `IndexSum` nodes in a form expression defines a reduction loop.

The general structure of a local assembly loop nest is given in Figure 5.11. Each local tensor entry is initialised to zero before the computation of its value takes place. Inside the quadrature loop, the subexpression values are computed. Some of these computations may be placed inside loops over reduction indices, depending on their placement in the expression tree. After subexpressions are computed, they are combined and added to the local tensor entry.

The particular loop nest that is constructed for the Helmholtz form is shown in Figure 5.12. In this example tetrahedral elements integrated with degree 2 quadrature are used. There is a single reduction index, `i1` which is used in the computation of the dot product.

```
for (int i=0; i<4; ++i) {
  for (int j=0; j<4; ++j) {
    A[i,j] = 0.0;
    for (int q=0; q<6; ++q) {
      for (int i1=0; i1<3; ++i1) {
      }
      A[i, j] += <expression>;
    }
  }
}
```

Figure 5.12: Local assembly loop nest for the Helmholtz form.

After the loop nest has been generated, expressions that compute the values of terms given in the form are generated. To achieve this, each partition in the expression is visited, and a subexpression that implements it is generated.

The `Product` and `Sum` nodes at the top of the tree may not fall inside any partition. These nodes at the root of the tree are visited last to create an expression that combines the other subexpressions from the partitions.

The outermost loop that any expression can be placed in is the quadrature loop, since all expressions must be evaluated for all entries of the local matrix and all quadrature points. In order to determine exactly where to place the expressions, the index sets for each partition are examined. For example, the left subexpression in Figure 5.10 has the index set $\{i_1\}$, so it should be placed inside the `d1` loop (this loop corresponds to the index i_1). The right partition has an empty index set, so should only be placed within the quadrature loop. The placing of expressions inside the loop nest is shown in Figure 5.13.

Once the structure of the loop nest is fully determined, expression generation can be performed. This is done by a visitor that descends through the expression tree in a depth-first manner. When terminals, such as test and trial functions, or coefficients are encountered, the corresponding low-level entity (such as a variable reference) is created and pushed onto a stack. When an operator is processed (such as a sum), the top two stack elements are popped and used as the operands. An entity representing this operator is generated with reference to these operands, and this new entity is placed on the stack. When the tree traversal terminates, the only remaining item on the stack is a single entity representing the entire expression.

```
for (int i=0; i<4; ++i) {
  for (int j=0; j<4; ++j) {
    A[i,j] = 0.0;
    for (int q=0; q<6; ++q) {
      SubExpr1 = <subexpression 1>
      for (int i1=0; i1<3; ++i1) {
        SubExpr2 = <subexpression 2>
      }
      A[i, j] += <expression combining SubExpr1 and SubExpr2>;
    }
  }
}
```

Figure 5.13: Local assembly loop nest for the Helmholtz form with expressions placed inside the correct loops.

```
for (int i=0; i<4; ++i) {
  for (int j=0; j<4; ++j) {
    A[i,j] = 0.0;
    for (int q=0; q<6; ++q) {
      SubExpr1 = test[i,q] * trial[j,q];
      for (int i1=0; i1<3; ++i1) {
        SubExpr2 = dtest[d1,i,q] * dtrial[d1,j,q];
      }
      A[i, j] += (SubExpr1 + SubExpr2) * detJ * w[q];
    }
  }
}
```

Figure 5.14: The final loop nest for the Helmholtz form. Values of test and trial functions are stored in `test` and `trial`. Values of the derivatives are stored in `dtest` and `dtrial`.

The expression generator is executed for each of the subexpressions, and the root of the tree. When generating the code for subexpressions, the visitor traverses all nodes in the expression tree until it reaches the leaves. When traversing from the root of the tree to generate the expression that combines all subexpressions, subexpression nodes are treated as terminals and a reference to the corresponding subexpression variable is generated when they are encountered. Since the root node is always the measure (`dx`), the quadrature terms are part of this expression, in particular the determinant of the Jacobian and an array of quadrature weights.

Figure 5.14 shows the loop nest for the Helmholtz form after expression generation. Note the local tensor entry expression combining the previous subexpressions and numerical quadrature terms.

5.5.2 Model Executor Generation

The Model Executor calls local and global assembly kernels and linear solves in the order specified in the Simulation DAG. For each `solve` that appears in the DAG, the left- and right-hand side forms that it uses are retrieved, and the following action is taken for each form:

- A call to `cudaMemset` that zeroes the global tensor is generated.
- The parameter list for the local assembly kernel is generated. Every parameter list includes the number of elements, a pointer to the local tensor, and the current size of the timestep. Additional parameters are pointers to the coefficient fields that are required by the kernel.
- A CUDA launch configuration is generated. A default configuration of 64 blocks with 128 threads each is used. Whilst it is possible to optimise the launch configuration to increase performance, this was not explored within the scope of the experiments for which MCFC was used.
- A unique name for the kernel is created.
- The kernel name, launch configuration, and parameter list are combined to create an invocation of the local assembly kernel.
- A call to an implementation of the Addto algorithm is generated.

Subsequently, a call to the solver that passes the global matrix and global vector is generated and inserted into the Model Executor.

5.5.3 Initialisation, Data Transfer, and Cleanup Code

The initialisation function is responsible for extracting simulation parameters from the Fluidity state (for example the number of elements and nodes), allocating memory on the GPU, and transferring initial conditions from the host to the device.

MCFC determines the fields that need to be initialised by examining the fields that are accessed in the Simulation DAG. The data of fields that are transferred is transformed to the layout described in Section 4.2.3. Space for the transformed fields is allocated on the GPU and data is transferred from the host to the device.

The initialisation function always transfers the coordinate field across, since it is required in all finite element simulations. It also allocates space for global matrix and vector data structures, and transfers the local-to-global mapping for the mesh.

The Data Transfer function copies all of the fields in the Simulation DAG that are assigned back into the state back to the host at the end of execution. Once the data has been transferred back to the host, it is transformed back to its original data layout so that it can be accessed correctly by Fluidity.

The cleanup code is responsible for freeing all memory on the GPU. It is generated by examining all the fields allocated in the Simulation DAG and creating calls that free memory for the structures that are used in the simulation.

5.6 Performance Comparison with DOLFIN

In order to evaluate the code generated by MCFC we compare it with the DOLFIN-generated code for an advection-diffusion problem. The UFL source code used for these experiments with some amendments for clarity is shown in Figure 5.15.

All benchmarks were executed on a machine with two 2.66GHz Intel Xeon X5650 (Westmere) CPUs that have 6 cores each, with Hyperthreading turned off, and 48GB of RAM installed. An NVidia GTX480 was used for running the CUDA code.

For linear algebra operations, DOLFIN was configured to use PETSc's conjugate gradient solver with Jacobi preconditioning. The CUDA code used a conjugate gradient solver with Jacobi preconditioning described in [MK09]. This combination of solver and preconditioner is

```
# Returns a Coefficient(FiniteElement('CG', 'triangle', 1))
t = state.scalar_fields['Tracer']

p=TrialFunction(t)
q=TestFunction(t)
diffusivity = 0.1

M=p*q*dx
adv_rhs = (q*t+dt*dot(grad(q),u)*t)*dx
t_adv = solve(M, adv_rhs)

d=-dt*diffusivity*dot(grad(q),grad(p))*dx
A=M-0.5*d
diff_rhs=action(M+0.5*d,t_adv)

tnew=solve(A,diff_rhs)
```

Figure 5.15: Advection-Diffusion equation source code.

chosen as it is the only one supported by the CUDA solver that was available at the time the experiments were performed.

DOLFIN Bzr revision number 6739 was used for these experiments. The DOLFIN version was run on 1-12 cores, using DOLFIN's built-in MPI parallelism. In order to avoid migration across NUMA domains, processes were pinned to a socket. The OpenMP backend was also evaluated, but it was not as performant as the MPI backend for this problem, and its benchmark results are omitted here. DOLFIN was configured to generate code using the tensor representation, since it was faster than the quadrature representation for this problem.

The GNU C++ compiler version 4.6.3 was used for compiling DOLFIN and its generated code, since it does not support the Intel compilers. The DOLFIN C++ compiler optimisation option was turned on, which sets the `-O2` flag when its generated code is compiled.

Form compiler optimisations were turned off. As described in [RK13], enabling form compiler optimisations causes FFC to generate code that evaluates local tensors with reduced accuracy. In the case of the forms used in these experiments, the reduced accuracy of the evaluation caused the solution to grow very large after a small number of timesteps.

There is only one configuration for the MCFC code, which is for it to run on a single GPU. The CUDA toolkit version 4.2 was used to compile the MCFC-generated code. The simulation

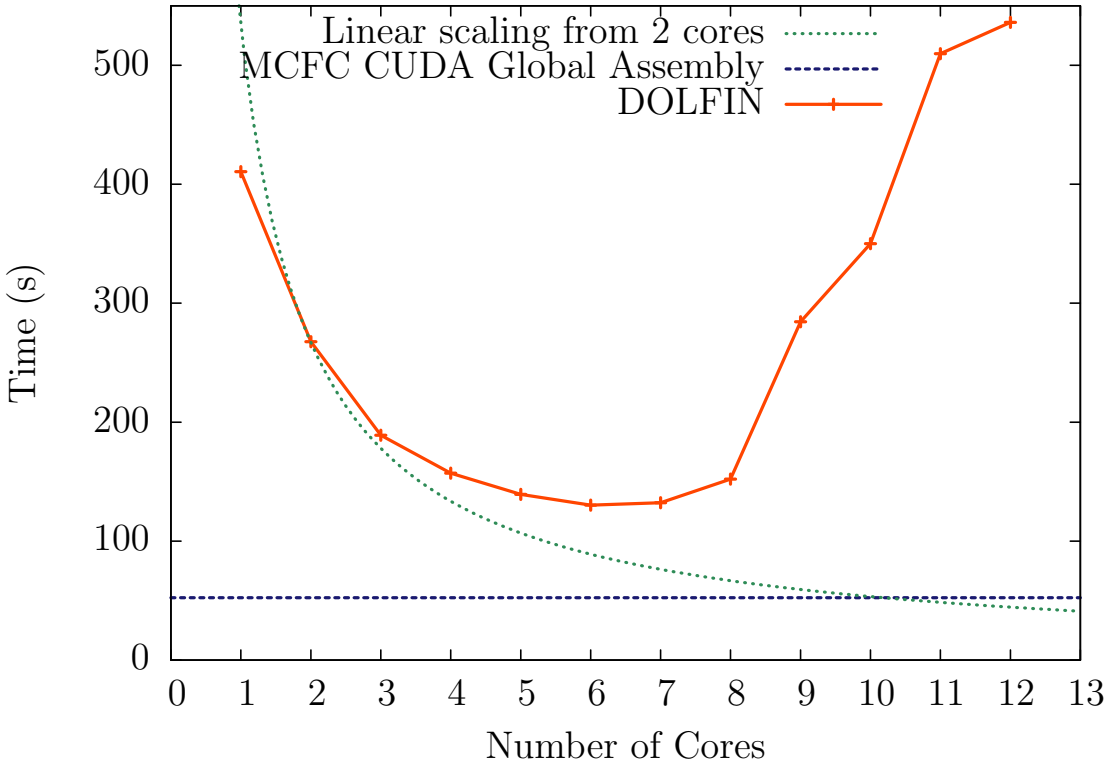


Figure 5.16: MCFC and DOLFIN comparison on a mesh with 172813 DOFs. The MCFC-generated CUDA code executes approximately 2.5 times faster than the closest DOLFIN run.

was run for 640 timesteps, which is sufficient to amortise the startup costs of both DOLFIN and MCFC.

Figure 5.16 compares the execution times of the DOLFIN and MCFC implementations for execution on a mesh with 172813 DOFs. This is the largest mesh that could be run with on the GPU due to memory constraints. Although this appears to be a relatively small mesh to use, it is the largest that could be used is because of the expanded data format used by the MCFC-generated code, and partly because there is some inefficiency in the MCFC code's memory management.

We see that the execution of the MCFC-generated code on CUDA is approximately 2.5× faster than the fastest DOLFIN run, which was on just 6 cores. If DOLFIN scaled linearly for this problem, then the performance of the DOLFIN and MCFC implementations should become equal at around 10 cores.

The scaling bottleneck preventing further improvement when adding cores was thought to be

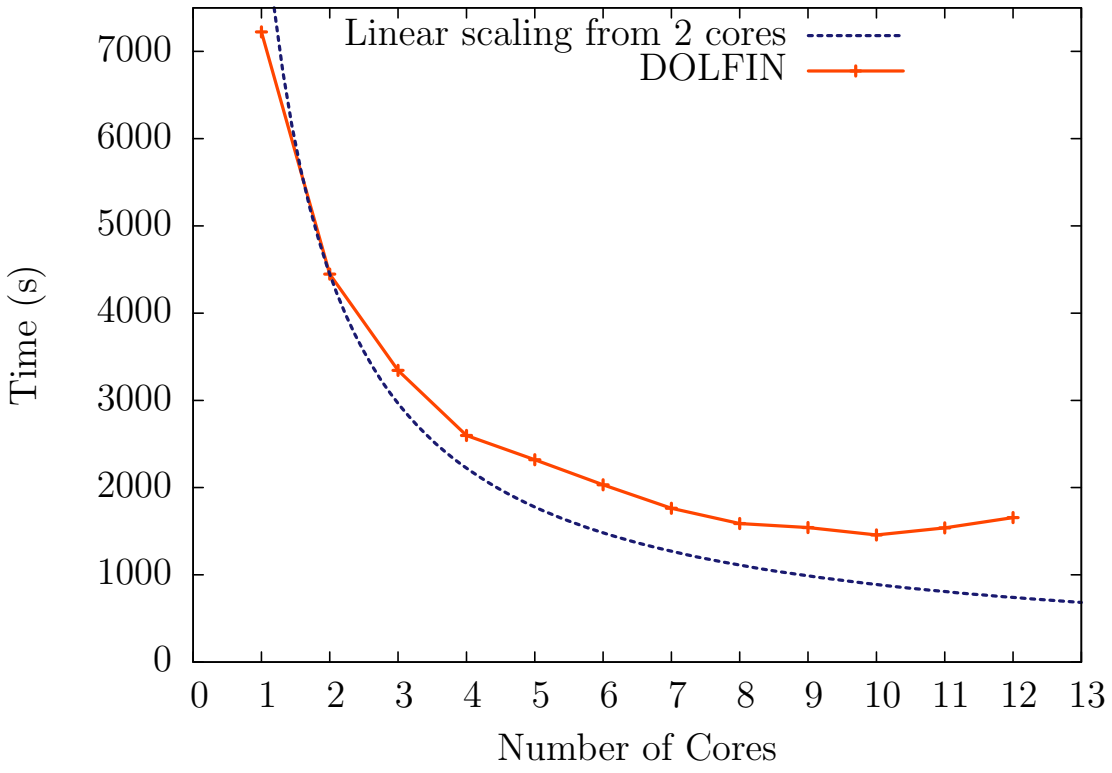


Figure 5.17: DOLFIN strong scaling on mesh with 1.7 million DOFs. Note the improvement in performance up to 10 cores.

due to the problem being too small to ensure that partitions are adequately sized. In order to test this hypothesis, we ran the DOLFIN implementation on a mesh with 1707280 DOFs in order to ensure that each process has a large enough partition size for efficient execution. Figure 5.17 shows the strong scaling of DOLFIN when executing with this mesh. We therefore conclude that the speedup of MCFC compared to DOLFIN in this benchmark is due to the small partition size necessitated by the size of the mesh used for the experiment. Given that DOLFIN scales close to linearly when partition sizes are large enough, we expect that DOLFIN-generated code running on 10 cores may provide equivalent performance to a single GPU running MCFC-generated code. More generally, we observe that DOLFIN exhibits good weak scaling but poor strong scaling for these particular experiments.

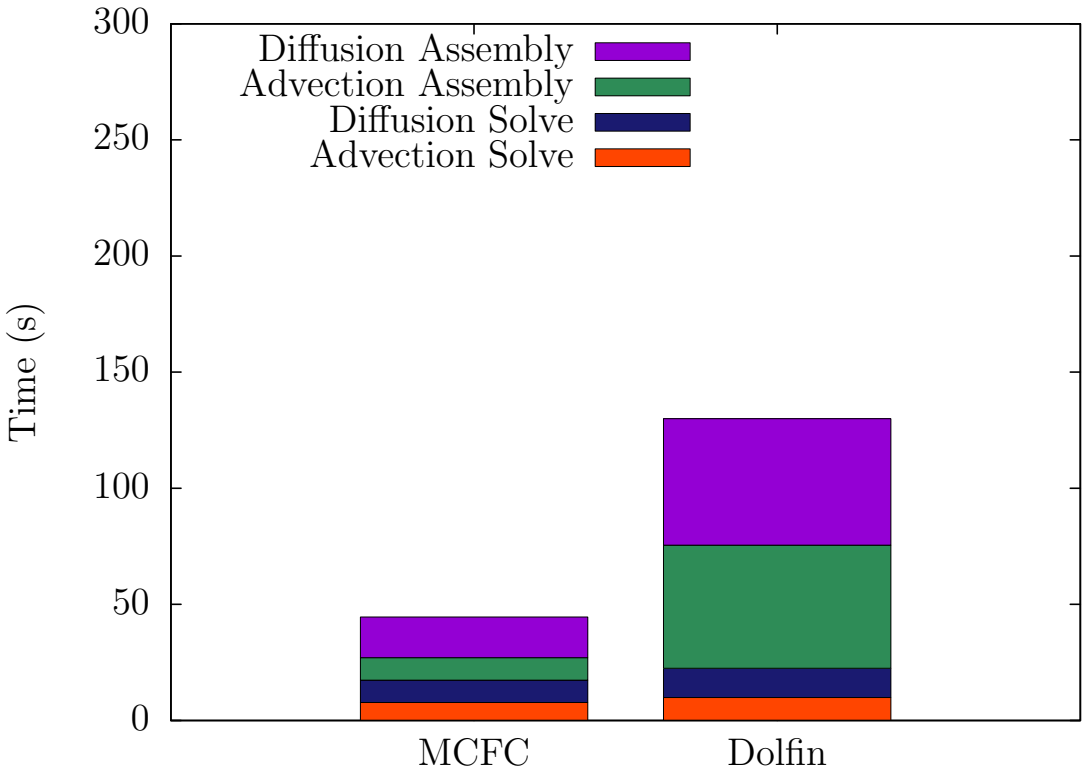


Figure 5.18: Breakdown of the execution time of MCFC and DOLFIN-generated code. The DOLFIN times are for execution on 6 cores, the fastest result. Note that the performance gain is the result of faster assembly, not faster linear system solving.

5.6.1 Execution profiles

In order to establish the source of the difference in execution times between the two implementations, we examine the execution profiles. These were recorded using DOLFIN's built-in timer infrastructure, and the MCFC timings are recorded by inserting calls to the C `time` function at appropriate points in the MCFC-generated code. Figure 5.18 shows a breakdown of the execution times of each implementation. The increase in performance of MCFC over DOLFIN is almost entirely attributable to better performance in the assembly phase. The CUDA-based solver is not highly-optimised, so it is possible that further performance increases in it would improve the performance of the MCFC implementation overall. However, for the comparison between the assembly phase only, the MCFC implementation is approximately 3.9× faster than DOLFIN.

A further breakdown of the execution times of the kernels in the MCFC code is given in Table

%	Kernel
28.7	Matrix addto
14.9	Diffusion matrix local assembly
7.1	Vector addto
4.1	Diffusion RHS
2.1	Advection RHS
0.5	Mass matrix local assembly
42.6	Solver kernels

Table 5.1: Percentage of execution times for each kernel in the MCFC implementation.

```
t = state.scalar_fields['Tracer']
M=p*q*dx

if spinup():
    rhs = discretisation1(t)
else:
    rhs = discretisation2(t)

tnew = solve(M, rhs)
```

Figure 5.19: Use of a conditional in a numerical method in MCFC. Assume the function `spinup` returns `True` early in the simulation and `False` after some time. Based on the spin-up status a different discretisation is selected.

5.1. We see that there is a significant overhead from the Addto kernels, and we conclude that the execution time of the MCFC-generated code could further be reduced by making use of the Local Matrix Approach as shown in the previous chapter. This enhancement to MCFC has not been implemented, since the focus of later work has been on the PyOP2 abstraction, which removes the responsibility for the global assembly algorithm from the form compiler.

5.7 OP2 Adaptation

In this section we discuss the conclusions that we draw from the development of MCFC about how OP2 should be modified to make it a more suitable parallel execution layer for performance-portable finite element assembly code.

5.7.1 Runtime Code Generation

Consider the code shown in Figure 5.19, which illustrates the use of a conditional based on a runtime value. This code is problematic for the static translation approach used in MCFC. In order to correctly translate the method, the code for both branches must be generated and the conditional must be implemented in the generated code. In order to perform this translation, MCFC would have to be substantially augmented to be able to analyse this code so that the Python semantics are captured in the generated code. It is not possible to execute the Python code and examine the state at the end of its execution to construct the generated code, since one of the branches will be taken during its execution.

Although it may be possible to perform translation for a limited subset of expressions such as conditionals, a robust mechanism would require a complete implementation of the semantics of Python in the code generator and the generated code. This requirement is unavoidable, because performing the code generation before executing any of the model requires the values of all the variables in the user's UFL to be known ahead-of-time, in order to have complete information to generate the code to assemble the specified forms.

It may be imagined that a "dry-run" execution of the UFL code would provide information about some of the values of the variables in the program. However, any variable value that depends on the result of a solve will be unknown. Because this would require code to be executed before it is generated, there is no static analysis that can provide enough information for correct, efficient and robust code generation of the whole model ahead of its execution.

A more straightforward approach to solving this problem would involve modifying OP2 so that it can generate code at runtime, when more information is available. In the case of the example from Figure 5.19, in the first timesteps where spin-up is occurring, the first branch will be taken and the code for discretisation 1 generated and called. When the spin-up phase ends, the branch for discretisation 2 will be entered and the code for this discretisation will be generated.

5.7.2 Iteration spaces

The partitioning and loop nest generation described earlier provide a method for reducing the size of the generated code in comparison to the code generated by FFC. However, this only goes part-way towards providing a mechanism for generating code that will execute efficiently on manycore architectures. It will also be necessary to map the workload onto available threads efficiently.

MCFC always assigns a single thread to each local tensor. This is a sufficiently efficient strategy for assembly with low-order elements that have been used in the benchmarks described in this chapter. However, as the order of basis functions increases, the working set of each thread will grow very large. The number of basis functions and quadrature points will increase, as will the number of entries in the local tensor. This problem is addressed for specific instances in some of the related work discussed in Section 3.2. For example, alternative mappings of the workload onto threads, such as the assignment of an entire thread block to a local matrix for sufficiently large local matrices.

The optimal mapping of the workload to threads will also depend heavily on the target device - for example, architectures with large caches will benefit from a coarse-grained assignment of the workload to threads. Therefore, in order for OP2 code to be performance-portable, the workload mapping should be entirely handled within its abstraction. We propose that this can be achieved by defining the iteration space of an OP2 kernel as the size of the local tensor, and performing the assignment of each element of the iteration space to available threads within the OP2 runtime system.

5.7.3 Linear Algebra

The CUDA conjugate gradient solver used by MCFC was sufficient for prototyping and carrying out the performance experiments described earlier. However, a wider set of choices of linear solver and preconditioner is necessary in order to solve a wide variety of problems. Implementations of linear algebra libraries for a range of multi-core and many-core devices will

also be required if performance-portability is to be obtained. For example, CUSP [BG12] and ViennaCL [WRS12] are linear algebra libraries that execute on manycore architectures, and there are various libraries available for multi-core machines.

In order to avoid duplicating the effort invested in these libraries, an interface that allows them to be used from OP2 should be developed. The OP2 API should provide methods for declaring the structure of matrices, assembling terms into these matrices, and solving a system of linear equations. This interface should invoke operations from the selected linear algebra library transparently to the code that uses its API.

5.8 Conclusions

In this chapter we have outlined a prototype form compiler implementation, MCFC, that obtains good performance for generated code executing on a single GPU in comparison to FEniCS/DOLFIN using FFC running on up to 12 cores. We conclude from the experiments with MCFC that there are three key changes that must be made to the design of OP2 to enable its use as a parallel execution layer in a performance-portable finite element toolchain. These are:

1. Switching from compile-time code generation to run-time code generation.

2. Provision of iteration spaces for finite element assembly.

3. The addition of interfaces to linear algebra libraries.

The expression partitioning and placement algorithms enable the generation of loop nests in order to minimise the size of local assembly kernels. These algorithms should be re-implemented in FFC in order to evaluate its performance in comparison with the present FFC algorithms, and to deliver them into production use. This implementation and evaluation is left for future work.

OP2 should be modified to use runtime code generation in order to robustly capture the semantics of finite element discretisations expressed in UFL. It should also provide a means for declaring the iteration space of local tensor assembly kernels in its API, so that it can implement an efficient mapping of work items to available threads. Finally, it should provide an API for linear algebra operations and an interface to linear algebra libraries that implement these operations. We discuss the process and results of making these modifications to OP2 in the following chapter.

Chapter 6

Designing PyOP2

6.1 Introduction

In this chapter we describe PyOP2, a re-implementation of the OP2 abstractions in Python that uses just-in-time compilation to generate code at runtime. Its design is influenced by the original OP2 API, and the recommendations laid out in the previous chapter. An API for matrix and vector assembly allows PyOP2 kernels to define local assembly operations. Global assembly and linear algebra operations are handled by linear algebra backends such as PETSc or CUSP, that are coupled through the PyOP2 linear algebra interface. PyOP2 supports generating CUDA, OpenMP and MPI code and provides an interface for adding additional targets.

We continue this chapter by summarising the requirements of PyOP2. We then describe its user interface, and explain the linear algebra interface and the concept of iteration spaces. Next, we describe the implementation of PyOP2 and conclude with a discussion of the PyOP2 design.

6.2 Specifications

The majority of the OP2 design is suitable for use as a parallel execution layer in a form compilation toolchain. The necessary changes to OP2 are:

Runtime code generation. As identified in Chapter 5 it is impossible to robustly transform numerical methods expressed in a combination of Python and UFL into an implementation expressed in a lower-level language or API. Runtime code generation is therefore necessary.

Vector maps. As described in Section 2.7.3, the OP2 requirement to pass a separate argument for each index in a map is problematic for the implementation of finite element local assembly kernels. The arity of maps can become large in finite element applications, which causes the available space for arguments to be exceeded on many-core architectures. To ensure that the number of kernel parameters is small enough, PyOP2 should pass all mapped values through a single pointer argument.

Linear algebra. An API for assembling matrices and vectors should be provided. It is unwise to implement linear algebra operations within the OP2 abstraction for two reasons. Firstly, OP2 does not allow the implementation of methods with a dependency between set elements, such as Gauss-Seidel, or methods that require the creation of new set elements, such as the ILU preconditioner. Secondly, any effort in this direction would be a duplication of the effort invested in developing the already-existing efficient and robust linear algebra libraries.

Iteration spaces. It is essential for PyOP2 to be able to control the mapping of local matrix entries to available threads in the assembly of matrices for linear algebra operations. This is to ensure that an efficient assignment of work items to threads is made, so that each thread has an appropriate working set size.

API Cleanup. The API should impose as little overhead as possible on the programmer. The specification of redundant information should be eliminated, and sensible defaults

```
Set(size, name=None)
Dat(dataset, dim, data=None, dtype=None, name=None)
Map(iterset, dataset, dim, values, name=None)
Kernel(code,name)
```

Figure 6.1: Constructors for data entities and kernels in PyOP2.

for parameters should be used where possible. In general, the API should also behave in the manner expected for a Python library, such that it follows the Principle of Least Astonishment [Sal09].

Rather than attempting to retrace the steps in the development of PyOP2, the remainder of this chapter will discuss the present implementation, with reference to how the above requirements are met.

6.3 User Interface Overview

Because the terminology and abstractions in OP2 have been retained, we will provide an overview of the PyOP2 interface and highlight its differences to the OP2 implementation. The API has been changed to make it more Pythonic, and as many parameters as possible have default values in order to avoid the repetition of common values in PyOP2 user code.

As in OP2, users can declare Sets, Dats, and Maps. The constructors for these entities are outlined in Figure 6.1. The names of entities can be passed for debugging purposes, but they are not mandatory. The constructors for Maps and Dats can accept any iterable type for their data or values, but the underlying data is held in NumPy arrays. If a NumPy array is passed in to the constructors, then it is used directly rather than being copied. This allows the use of data that PyOP2 does not own, so that interfaces to non-PyOP2 code can be created.

PyOP2 kernels follow the same specification as OP2 kernels, and are embedded in the user's code as C strings. For the restricted set of semantics that are allowed in kernels, it is likely that a translation from Python functions into C kernels is possible. This translation has not

```
par_loop(kernel, iteration_space, *args)
Set.__call__(self, *dims)
Dat.__call__(self, path, access)
```

Figure 6.2: PyOP2 parallel loop API. The `iteration_space` argument is constructed by calling a `Set`. Further arguments are constructed by calling a `Dat`.

yet been implemented since the primary use for PyOP2 has been to accept input from a form compiler that already generates C kernels.

Debugging of kernel code can be accomplished using the GNU Debugger, `gdb`, when running with the sequential backend (the backends will be outlined in Section 6.5.2). A traceback up to the beginning of the execution of generated code can be obtained, and it is possible to examine the data structures passed to kernels as normal.

A PyOP2 kernel object is constructed from its code and the name of the kernel. The kernel name must be the same as the one in its source code. This is required so that PyOP2 can invoke the kernel function correctly.

The parallel loop API is outlined in Figure 6.2. Parallel loops are invoked with a call to `par_loop`, where the first argument is a `Kernel`. The next argument is an iteration space, which replaces the set argument used in OP2. The concept of an iteration space will be discussed in Section 6.4.2. The remaining arguments to a parallel loop are of the `Arg` type. The user does not instantiate these types directly, but they are created by calling `Dat` or `Mat` objects. The provision of the `__call__` methods provides a natural syntax for parallel loop arguments, an example of which will be seen in the following section. The call method for `Dat` objects accepts a path, which is an optionally-indexed `Map`, and an access descriptor, which is one of `READ`, `WRITE`, `RW`, `INC`, `MAX`, or `MIN`, with the same meaning as in OP2. If a `Map` is indexed, this index is used to fetch a single `Map` value for each iteration set element that is used to index into the dataset. If no index is passed, then all map values for a single element of the iteration set are gathered, and PyOP2 assembles a list of pointers to each value in the dataset that is passed to the user's kernel.

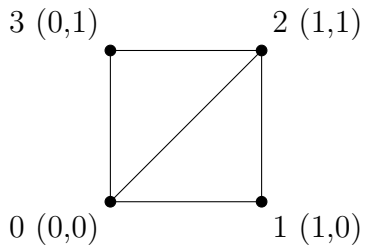


Figure 6.3: A two-cell, four-node mesh.

```
from pyop2 import op2
op2.init(backend='sequential')

# Two cells and four vertices
cells       = op2.Set(2)
vertices    = op2.Set(4)
# Three vertices per cell
cell_vertex = op2.Map(cells, vertices, 3, [ (0, 1, 2), (2, 3, 0) ] )

# Data on sets
vertex_coords = op2.Dat(vertices, 2,
                        [ (0.0, 0.0), (1.0, 0.0), (1.0, 1.0), (0.0, 1.0) ] )
cell_centre   = op2.Dat(cells, 2,
                        [ (0.0, 0.0), (0.0, 0.0) ] )
cell_mass     = op2.Dat(cells, 1,
                        [ 0.0, 0.0 ] )
```

Figure 6.4: Initialisation of PyOP2 and data declarations for the two-element mesh. Although this example uses lists for initial data, NumPy arrays can be passed directly to `Dat` and `Map` constructors.

6.3.1 API Example

In order to illustrate the use of the API, we provide a small example that declares a mesh and uses a parallel loop to compute the mass and centre of cells. The mesh is shown in Figure 6.3. PyOP2 is initialised and a representation of this mesh is constructed in the code shown in Figure 6.4. In the example the `Map` and `Dat` values have a shape that matches the structure of the data. It is also possible to pass in flat arrays of values, and in this case PyOP2 reshapes the data as necessary.

The definition of a kernel that computes both the mass and centre is given in Figure 6.5. An alternative implementation may use separate kernels for these operations. The coordinate parameter, `x` is accessed indirectly so its values are passed as a double pointer by PyOP2 (shown as `*x[2]` here to clarify the extent of the data). The `centre` and `mass` parameters are accessed

```
mass_centre = op2.Kernel("""
void mass_centre(double *x[2], double centre[2], double mass[1])
{
  centre[0] = (x[0][0] + x[1][0] + x[2][0]) / 3.0;
  centre[1] = (x[0][1] + x[1][1] + x[2][1]) / 3.0;
  mass[0] = abs(x[0][0]*(x[1][1]-x[2][1]) + x[1][0]*(x[2][1]-x[0][1]) +
                x[2][0]*(x[0][1]-x[1][1]) ) / 2.0;
}""", "mass_centre")

op2.par_loop(mass_centre, cells,
             vertex_coords(cell_vertex, op2.READ),
             cell_centre(op2.IdentityMap, op2.WRITE),
             cell_mass(op2.IdentityMap, op2.WRITE))
```

Figure 6.5: Declaration and invocation of a kernel that computes the mass and centre of cells.

directly so a single pointer to their data for the current iteration set element is passed by PyOP2.

The invocation of the kernel specifies iteration over cells. The final three arguments passed to the parallel loop are generated by calling the `Dat` objects representing the data to be passed. Because the iteration is over cells but `vertex_coords` is defined on vertices, it must be accessed indirectly through a map from cells to vertices. In this case the `cell_vertex` map is used. Coordinates are only read and not written, so the `READ` access descriptor is used. The cell centre and mass are both defined on the set of cells, so the identity map can be used to access them directly. The `WRITE` access descriptor is provided since they are written to without conflict.

6.4 Linear Algebra Interface

The PyOP2 interface for constructing matrices is loosely modelled on the interface provided by PETSc. In order to construct a matrix, a matrix sparsity must first be created. A sparsity stores information about the non-zero structure of a matrix. A matrix object, which holds the matrix values, can then be declared on the sparsity. Multiple matrices can share a single sparsity.

Since the sparsity of matrices in finite element methods is related to the structure of the mesh, sparsities are declared using maps. The sparsity structure is derived from a pair of maps, the

```
Sparsity(maps, dims, name=None)
Mat(sparsity, dtype=None, name=None
```

Figure 6.6: Constructors for `Sparsity` and `Mat` objects.

```
dofmap = op2.Map(cells, vertices, 3, [ (0, 1, 2), (2, 3, 0) ] )
sparsity = op2.Sparsity((dofmap, dofmap), 1)
mass = op2.Mat(sparsity, "mass")
```

Figure 6.7: Declaration of matrix data structure for a mass matrix on the example mesh. The `dofmap` values coincide with the cell vertices - this is the case for P1 Lagrange elements but in general they will differ. The `dofmap` is passed for both the rows and columns since the test and trial spaces have the same basis. This need not be true in general.

row map and the *column map*. The product of these maps is used to form the matrix sparsity pattern. It is not possible to arbitrarily declare the sparsity pattern of a matrix, and this is not required in the implementation of finite element methods.

The constructors for `Sparsity` and `Mat` objects are outlined in Figure 6.6. The `Sparsity` constructor accepts a parameter for specifying the dimension of an underlying field. This allows the construction of sparsities that can be used for assembling vector fields as well as scalar fields.

A continuation of the example code from earlier that declares a mass matrix for the mesh is given in Figure 6.7. Although the map from cells to degrees of freedom coincides with the map to vertices, this is not the case in general so a separate `Map` called `dofmap` has been declared to reflect the steps that will be required in the general case.

6.4.1 Matrix Assembly

As with `Dat` values, the values of `Mat` objects are assigned into during execution of kernels. When a kernel that writes to a matrix is invoked, it computes a local matrix value. The value is added in to the global matrix by PyOP2, which obtains the local-to-global mapping from the `Sparsity` on which the `Mat` is declared. The Sparsity holds the row map and column map that were used to produce the local-to-global mapping.

One possible way of extending the OP2 kernel API to support matrix assembly would involve

specifying that user kernels should contain the code for the assembly of an entire local matrix. This would fit naturally into the existing API, since one local matrix will be constructed for each member of the iteration set. However, it will be difficult to assign multiple threads to the execution of user kernels written in this way, since it will be necessary to analyse kernels in this form in order to build a model of their iteration space, and to ensure that there are no hazards that would enforce an ordering on iterations which would prevent execution in parallel.

The alternative to this approach, which we have chosen to implement in PyOP2, involves stipulating that user kernels assemble a single element of the local matrix. The assembly of the whole local matrix is achieved by the generation of a loop over local matrix entries, which the user kernel is inlined into. The dimensions of the local matrix are specified along with the iteration set in the parallel loop invocation, to construct the *iteration space*. A two-dimensional iteration space is constructed for local matrix assembly, and a one-dimensional iteration space can be constructed for local vector assembly. Calling the set with two integers specifies the size of the local matrix, or in the case of vector assembly a single integer is used.

An example of a matrix assembly kernel and its invocation are given in Figure 6.8. The kernel argument for the matrix is passed as a pointer to the data for a single element of the matrix. In the case where a scalar field is being assembled, this is a 1×1 array. If a 2D vector field were being assembled, the parameter `double A[2][2]` would be passed instead.

The size of the iteration space is indicated by `cells(3,3)`. For each set element, instead of invoking the kernel once, the kernel is invoked $3 \times 3 = 9$ times, once for each local matrix element. Since the computation for each local matrix element depends on its location in the local matrix, the current point in the iteration space is passed as two integer parameters that are at the end of the parameter list (`i` and `j`).

It is necessary for PyOP2 to be made aware of which local matrix element is assigned to at each point in the iteration space. This is indicated by indexing the maps that are used to access the matrix with the special indices `op2.i`. The first iteration space parameter is indicated with the use of `op2.i[0]` and the second with the use of `op2.i[1]`.

```
code = """
void mass(double A[1][1], double *x[2], int i, int j)
{
    // Compute Jacobian and its determinant
    const double J_00 = x[1][0] - x[0][0];
    const double J_01 = x[2][0] - x[0][0];
    const double J_10 = x[1][1] - x[0][1];
    const double J_11 = x[2][1] - x[0][1];
    const double det = fabs(J_00*J_11 - J_01*J_10);

    // Quadrature weights and basis function values
    const double W3[3] = \
     {0.166666666666667, 0.166666666666667, 0.166666666666667};
    const double FE0[3][3] = \
    {{0.666666666666667, 0.166666666666667, 0.166666666666667},
     {0.166666666666667, 0.166666666666667, 0.666666666666667},
     {0.166666666666667, 0.666666666666667, 0.166666666666667}};

    // Loop over quadrature points and compute local matrix entry value
    for (unsigned int ip = 0; ip < 3; ip++)
      A[0][0] += FE0[ip][i]*FE0[ip][j]*W3[ip]*det;
}"""
mass_kernel = op2.Kernel(code, "mass")

op2.par_loop(mass, cells(3,3),
             mat((elem_node[op2.i[0]], elem_node[op2.i[1]]), op2.INC),
             coords(elem_node, op2.READ))
```

Figure 6.8: Mass matrix kernel declaration and invocation. The iteration space is constructed by passing the local matrix size to the iteration set call. Indexing into the iteration space is accomplished through the special indices `op2.i`.

```
op2.par_loop(kernel, cells(12,12),
             mat(cell_dof[op2.i[0]], cell_dof[op2.i[1]]),
             *args)
```

Figure 6.9: Parallel loop for matrix assembly with a 12×12 iteration space.

```
for (ele=TID; ele+=NT; ele<n)
  for (i=0; i<12; i++)
    for (j=0; j<12; j++)
      user_kernel(..., i, j)
      addto(matrix, i, j, ele)
```

Figure 6.10: Code generated when each local matrix is assigned to a single thread.

After each invocation of a kernel that assembles a matrix entry, the value(s) computed by the kernel are added into the global matrix by PyOP2. The current iteration space values are used to index into the relevant maps in order to determine the global matrix entry that the value is added into.

6.4.2 Local Assembly Code Generation

To illustrate how the iteration space API provides PyOP2 with the opportunity to change the assignment of work to threads depending on the target architecture, we consider the assembly of local matrices with a 12×12 iteration space. This iteration space is sufficiently large that assigning the entire space to a single thread is unlikely to be an optimal strategy on a many-core platform. The parallel loop invocation for this example is shown in Figure 6.9.

The code generated when using a simple strategy in which the entire local matrix is assembled by a single thread is shown in Figure 6.10. This strategy may work well on a multi-core architecture with a large cache.

An alternative strategy where the iteration space is split into 4×4 tiles could result in the code shown in Figure 6.11. In this strategy, nine threads are assigned to compute the values of each matrix. This implementation may be more efficient on many-core architecture. Alternatively, this form of the code may be transformed so that it can exploit SIMD vectorisation on a multi-core architecture.

```
user_kernel(..., int i, int j) { A[i,j] += ...; }

for (ele=TID/3; ele+=NT/3; ele<n)
  patch_i = TID%3;
  patch_j = (TID%9)/3;
  for (i=0; i<4; i++)
    for (j=0; j<4; j++)
      ki = patch_i*4 + i; kj = patch_j*4 + j;
      user_kernel(..., ki, kj);
      addto(matrix, ki, kj, ele)
```

Figure 6.11: Code generated when each local matrix is computed by 9 threads in total.

Figure 6.12: Iteration space for a 12×12 local matrix. Left: Entire iteration space assigned to a single thread. Right: Iteration space divided into tiles for execution using CPU vector instructions or a GPU.

It is likely that the optimal implementation would have to be determined experimentally. The key point which we draw attention to is that all of these implementations can be automatically generated by PyOP2 without any change to the user kernel or interaction from a user or higher-level code generator.

A graphical representation of the assignment of threads to elements in the iteration space is shown for the two strategies discussed, in Figure 6.12.

```
Mat.zero_rows(self, rows, diag_val)
Mat.zero(self)
```

Figure 6.13: Interface for zeroing either a row of a matrix or the entire matrix. Zeroing rows allows the value of the diagonal entries of the zeroed rows to be specified and is used for setting Dirichlet boundary conditions.

Parameter	Default value
`linear_solver`	`'cg'`
`preconditioner`	`'jacobi'`
`relative_tolerance`	1.0×10^{-7}
`absolute_tolerance`	1.0×10^{-50}
`divergence_tolerance`	1.0×10^{4}
`maximum_iterations`	1000
`error_on_nonconvergence`	`True`
`gmres_restart`	30

Table 6.1: Supported PyOP2 solver parameters and default values.

6.4.3 Setting Matrix Entries Directly

Although arbitrary modification of matrix entries is prohibited, an interface for zeroing the off-diagonal entries of particular rows in a matrix is provided for the purpose of implementing Dirichlet boundary conditions. This interface, which is modelled on the PETSc `MatZeroRows` function, is shown in Figure 6.13. The `zero_rows` function takes a list of rows and a value for the diagonal term. It is usual for the diagonal value to be 1.0 when implementing a Dirichlet boundary condition.

Another function is provided for setting all the entries of a matrix to zero. This allows matrices to be reused, as opposed to forcing the construction of new `Mat` objects. This method requires no parameters, and operates on the matrix on which it is called.

6.4.4 Solving

The PyOP2 solver interface allows a matrix and a vector to be used to form a linear system in order to solve for a new vector. In the PyOP2 context, vectors are represented by `Dat` objects.

The configuration of a linear solver is achieved by creating a `Solver` object and passing a `dict` of parameters. The parameters that are presently supported are a subset of the PETSc parameters, and have the same default values as the PETSc versions. An overview of supported parameters is given in Table 6.1. Once the solver has been configured, the `solve` method can be called. Figure 6.14 gives an overview of the `Solver` API.

```
Solver(parameters, **kwargs)
Solver.solve(A, x, b)
```

Figure 6.14: PyOP2 solver object interface. A `dict` of parameters or individual keyword arguments for overriding default parameters may be passed to the constructor.

6.5 Implementation

In this section we describe the PyOP2 implementation, and draw attention to the fact that its development has largely been a team effort between the author of this thesis and several other contributors.

An overview of the components in PyOP2 implementation is given in Figure 6.15. The API is provided by the *Runtime Core* component, which also contains the interface to linear algebra libraries. The core provides an API for the implementation of target-specific backends. At present, backends for sequential C code, C with OpenMP, CUDA, and OpenCL code are fully supported. PyOP2 also supports MPI with the CPU backends, and overlaps communication and computation where possible.

Rather than providing an exhaustive description of all the components in PyOP2 and their interactions, we will consider the salient points of its implementation, and those in which it differs from OP2.

6.5.1 Plan Caching

Prior to kernel invocation, a *plan* is generated for its execution. Plans in PyOP2 perform the same task as in OP2, which is described in Section 2.7.3. Since plan construction can be expensive, caching of plans is implemented to ensure that construction is only performed once. When a parallel loop is invoked, the cache key for the loop is generated, and used to check the cache. If the key exists in the plan cache, the cached plan is retrieved. Otherwise, the plan construction is invoked and the resulting plan is added to the cache. Subsequently, parallel loop execution proceeds using the plan.

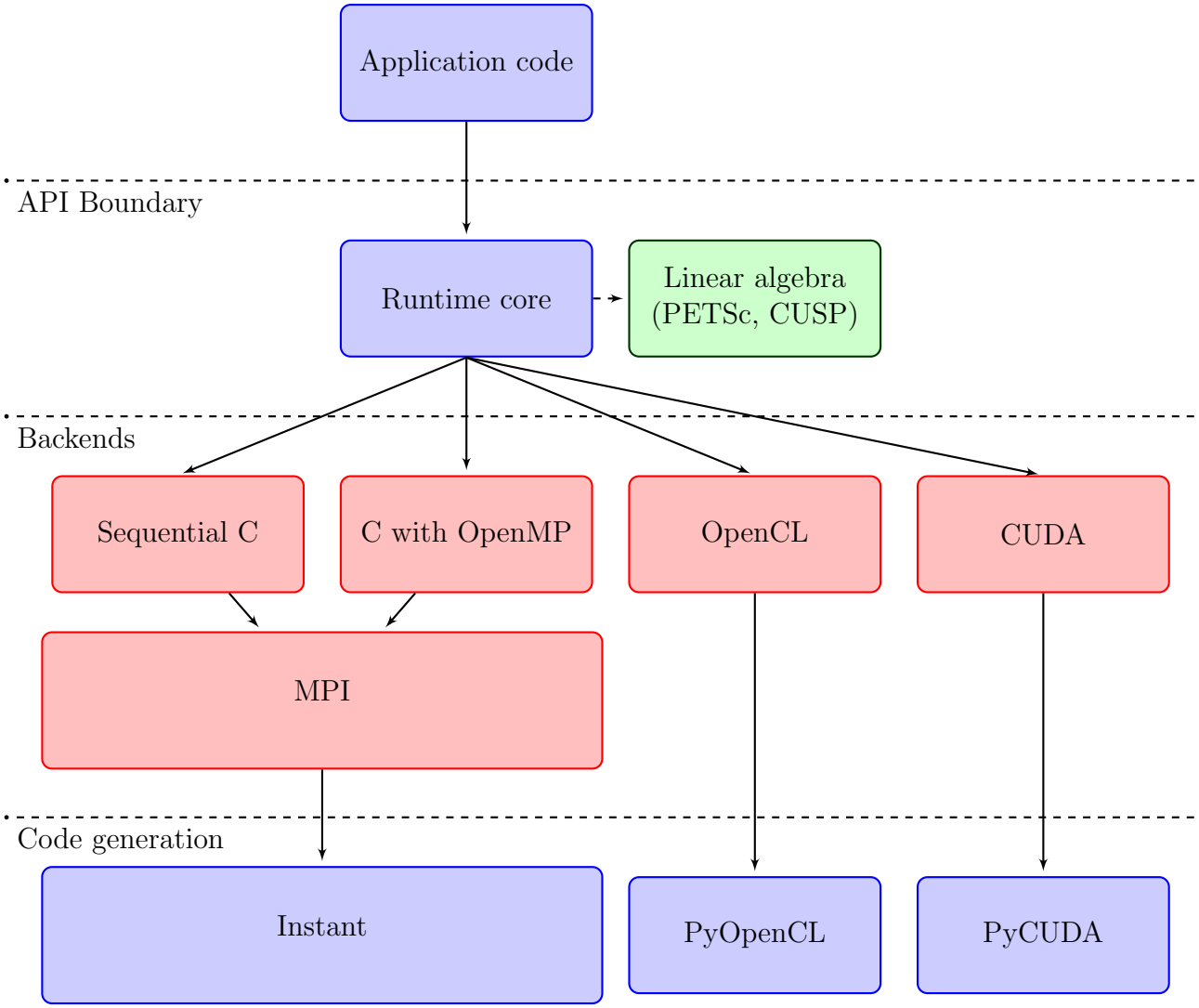


Figure 6.15: An overview of the PyOP2 architecture.

Class	Core functionality
`Set`	Recording set size.
`Map`	Recording iterset and dataset.
`Dat`	Recording dataset, type, and dimension.
`Sparsity`	Sparsity pattern construction, recording rowmaps and colmaps.
`Mat`	Recording sparsity.
`Kernel`	Recording kernel code and name.
`par_loop`	No responsibility.
`ParLoop`	Plan cache key generation.
`Solver`	Recording solver parameters.

Table 6.2: Responsibilities of data structures defined in the runtime core. All classes also perform validation of arguments passed to them. Because data storage is different for every backend, the Runtime Core does not hold any `Dat` or `Map` values, only properties of the data.

The cache key is constructed based on the data accessed by a parallel loop. This data is defined by:

- the shape of the iteration space,
- the `Dat` and `Mat` objects passed as arguments,
- the `Maps` through which this data is accessed,
- and the access specifiers (READ, WRITE, etc.).

These items are safe to use as the key for the plan cache because the structure of PyOP2 `Dats`, `Maps` and `Mats` is immutable - only the data values change as execution proceeds.

6.5.2 Multiple Backend Support

Backends in PyOP2 are implemented by inheriting from classes for each of the core entities in PyOP2. The runtime core implements functionality common to all backends in these classes. An outline of the responsibilities handled by each of these core classes is given in Table 6.2. It is not always necessary to subclass `Kernel` and `Set` when implementing a backend, but it is permitted, and may be convenient.

```
void addto_scalar(Mat mat, const void *value, int row, int col, int insert)
void addto_vector(Mat mat, const void *values,
                  int nrows, const int *irows,
                  int ncols, const int *icols, int insert)
```

Figure 6.16: Stub functions for matrix assembly for CPU-based linear algebra libraries.

A backend's implementation of the `Map` and `Dat` classes is responsible for the allocation and management of memory to hold the data that underlies them. It is also responsible for transferring data between the host and the backend target device if and when necessary.

Implementation of the `par_loop` function is mandatory. The backend should generate, compile, and dynamically link the code for parallel loops, and control their execution on the target device. The `ParLoop` class may be subclassed as part of the implementation of this functionality. The `generate_code` and `compute` methods of the `ParLoop` class are placeholders for some of the functionality of a parallel loop.

The implementation of a linear algebra backend is accomplished by overriding the `Mat` and `Solver` classes. The `Mat` implementation must construct a matrix representation using the underlying library, and must provide an interface for assembling into terms of the matrix, that can be called by the generated code. For linear algebra solvers that run on the CPU, this is achieved by implementing the stub functions shown in Figure 6.16. For linear algebra backends that execute on other devices, there is presently no standard interface defined for matrix assembly since the only non-CPU linear algebra library supported is CUSP.

6.5.3 CPU Backends

The CPU backends can generate either sequential C code or C with OpenMP directives for shared-memory parallel execution. In order to implement the traversal over set elements, the backend generates a loop over set elements that contains a call to the kernel code.

Figure 6.17 gives an example of the code that is generated for the example `mass_centre` kernel presented earlier. The generated wrapper accepts Python objects as parameters and extracts the necessary values from them. This includes the bounds of the loop and pointers to the

```
void mass_centre(double *x[2], double centre[2], double mass[1])
{
  centre[0] = (x[0][0] + x[1][0] + x[2][0]) / 3.0;
  centre[1] = (x[0][1] + x[1][1] + x[2][1]) / 3.0;
  mass[0] = abs(x[0][0]*(x[1][1]-x[2][1]) + x[1][0]*(x[2][1]-x[0][1]) +
                x[2][0]*(x[0][1]-x[1][1]) ) / 2.0;
}

void wrap_mass_centre__(PyObject *_start, PyObject *_end,
                        PyObject *_dat_0, PyObject *_dat_0_map,
                        PyObject *_dat_1,
                        PyObject *_dat_2 )
{
  int start = (int)PyInt_AsLong(_start);
  int end = (int)PyInt_AsLong(_end);

  double *dat_0 = (double *)(((PyArrayObject *)_dat_0)->data);
  int *dat_0_map = (int *)(((PyArrayObject *)_dat_0_map)->data);
  double *dat_0_vec[3];
  double *dat_1 = (double *)(((PyArrayObject *)_dat_1)->data);
  double *dat_2 = (double *)(((PyArrayObject *)_dat_2)->data);

  for ( int i = start; i < end; i++ ) {
    dat_0_vec[0] = dat_0 + dat_0_map[i * 3 + 0] * 2;
    dat_0_vec[1] = dat_0 + dat_0_map[i * 3 + 1] * 2;
    dat_0_vec[2] = dat_0 + dat_0_map[i * 3 + 2] * 2;

    mass_centre(dat_0_vec, dat_1 + i * 2, dat_2 + i * 1);
  }
}
```

Figure 6.17: Sequential backend-generated code for the `mass_centre` kernel. Iteration set bounds are passed in the first two arguments and used to control the loop over set elements. The indirectly-accessed argument `x` (the coordinate field) has its values gathered through the appropriate map before invocation of the user's kernel.

values of maps and data. The coordinate field is accessed indirectly through a map that is not indexed. Because the map is not indexed, all of the coordinate field values for each element of the iteration set are gathered into a temporary array. This temporary array of pointers, `dat_0_vec`, is populated by accessing the coordinate field map to compute offsets to the correct data items for the current element. The user kernel can then access the correct elements in the coordinate data array by subscripting this array of pointers.

The CPU backends use the Instant library [IMWe12] for compiling the code and linking it back into the Python interpreter. Instant handles compilation, linking, and caching of the generated code so these concerns do not need to be addressed by the CPU backends.

At present the CPU backends make use of the PETSc linear algebra backend. This is a sensible choice because the matrix assembly wrappers can be called directly by the generated code. It would be possible to adapt the CPU backends to use the CUSP solver, but this would involve performing matrix assembly on the host, then transferring the matrix to a GPU for solving, and another transfer to retrieve the solution. In the circumstances where a CUDA-capable GPU is available, it is likely to be more efficient to use the CUDA backend rather than a combination of CPU backend and GPU solver.

The operation of the OpenMP backend is similar in principle to that of the sequential backend. Since it executes loops in parallel, it requires an execution plan to be constructed. Instead of using a flat loop over elements, the outer loop of wrapper functions generated by the OpenMP backend iterates over colours instead of iteration set elements. The code generated by the OpenMP backend for the mass kernel is shown in Figure 6.18.

The inner loop iterates in parallel over partitions within one colour, assigning one thread to each partition. This strategy avoids the need for synchronisation within a partition. The temporary space for vectors of pointers to data elements is expanded so that each thread can maintain a private copy of the pointers.

On NUMA architectures, such as multi-socket x86 machines, it is conceivable that performance of the OpenMP backend may be reduced if threads make a large number of accesses to data

```
void wrap_mass_centre__(PyObject *_dat_0, PyObject *_dat_0_map,
                        PyObject *_dat_1, PyObject *_dat_2 ,
                        PyObject* _part_size, PyObject* _ncolors,
                        PyObject* _blkmap, PyObject* _ncolblk,
                        PyObject* _nelems)
{
  int part_size = (int)PyInt_AsLong(_part_size);
  int ncolors = (int)PyInt_AsLong(_ncolors);
  int* blkmap = (int *)(((PyArrayObject *)_blkmap)->data);
  int* ncolblk = (int *)(((PyArrayObject *)_ncolblk)->data);
  int* nelems = (int *)(((PyArrayObject *)_nelems)->data);

  double *dat_0 = (double *)(((PyArrayObject *)_dat_0)->data);
  int *dat_0_map = (int *)(((PyArrayObject *)_dat_0_map)->data);
  double *dat_0_vec[32][3];
  double *dat_1 = (double *)(((PyArrayObject *)_dat_1)->data);
  double *dat_2 = (double *)(((PyArrayObject *)_dat_2)->data);

  int nthread = omp_get_max_threads();

  int boffset = 0;
  for ( int __col  = 0; __col < ncolors; __col++ ) {
    int nblocks = ncolblk[__col];

    #pragma omp parallel default(shared)
    {
      int tid = omp_get_thread_num();

      #pragma omp for schedule(static)
      for ( int __b = boffset; __b < (boffset + nblocks); __b++ ) {
        int bid = blkmap[__b];
        int nelem = nelems[bid];
        int efirst = bid * part_size;
        for (int i = efirst; i < (efirst + nelem); i++ ) {
          dat_0_vec[tid][0] = dat_0 + dat_0_map[i * 3 + 0] * 2;
          dat_0_vec[tid][1] = dat_0 + dat_0_map[i * 3 + 1] * 2;
          dat_0_vec[tid][2] = dat_0 + dat_0_map[i * 3 + 2] * 2;

          mass_centre(dat_0_vec[tid], dat_1 + i * 2, dat_2 + i * 1);
        }
      }
    }

    boffset += nblocks;
  }
}
```

Figure 6.18: Wrapper code for the `mass_centre` kernel generated by the OpenMP backend. Execution over partitions of the same colour is performed by the outer loop. The inner loop assigns each partition to a single thread. Temporary space for the coordinate field is expanded so that each thread maintains a private copy.

outside the NUMA domain of the core that they are running on. This may be mitigated by pinning OpenMP threads to particular sockets, and implementing a first-touch strategy to keep the majority of memory accesses within a single NUMA domain. The implementation of these optimisations in PyOP2 remains for future work.

6.5.4 Device Backends

In this section we describe the CUDA and OpenCL backends, whose operation are generally similar. We will focus the discussion on the CUDA backend since it is usable in a complete finite element toolchain, whereas the OpenCL backend lacks suitable linear algebra support at present.

Figure 6.19 gives an example of the kernel generated by the CUDA backend for the `mass_centre` parallel loop from earlier. Generated kernels execute over the partitions in a single colour. The PyOP2 runtime launches the kernel once for each partition colour, as shown in Figure 6.20. When the execution of the kernel begins, a single thread in each block retrieves the size of the partition and a pointer to the map values for the partition. Indirectly-accessed data is then staged into shared memory.

In the `mass_centre` example, there is no indirectly-accessed data that is written to. When indirectly accessed data is written, a further loop after the user kernel invocation loop is generated, which copies the data back to global memory from shared memory.

Because GPUs have a separate memory space to that of the host, it is necessary for the device backends to ensure that there are correct copies of data on the host and the device when they are required. Data can be in one of the following four states:

Device unallocated. This is the initial state when a `Dat` is created. An array containing the data exists on the host only but not on the device.

Host. Space for the data exists on both the host and the device. The version on the host is considered the current version, and may contain different values to the version on the

```
__global__ void
__mass_centre_stub(int set_size,
                   double *dat_0, double *dat_1, double *dat_2,
                   int *ind_map, short *loc_map, int *ind_sizes,
                   int *ind_offs, int block_offset,
                   int *blkmap, int *offset, int *nelems,
                   int *nthrcol, int *thrcol, int nblocks)
{
  extern __shared__ char shared[];
  __shared__ int *dat_0_map, dat_0_size;
  __shared__ double * dat_0_shared;
  __shared__ int nelem, offset_b;

  double *dat_0_vec[3];

  if (blockIdx.x + blockIdx.y * gridDim.x >= nblocks) return;
  if (threadIdx.x == 0) {
    int blockId = blkmap[blockIdx.x + blockIdx.y * gridDim.x +
                         block_offset];
    nelem = nelems[blockId];
    offset_b = offset[blockId];

    dat_0_size = ind_sizes[0 + blockId * 1];
    dat_0_map = &ind_map[0 * set_size] + ind_offs[0 + blockId * 1];
    dat_0_shared = (double *) &shared[0];
  }

  // Copy into shared memory
  __syncthreads();
  for ( int idx = threadIdx.x; idx < dat_0_size * 2; idx += blockDim.x )
    dat_0_shared[idx] = dat_0[idx % 2 + dat_0_map[idx / 2] * 2];
  __syncthreads();

  for ( int idx = threadIdx.x; idx < nelem; idx += blockDim.x )
  {
    dat_0_vec[0] = dat_0_shared + loc_map[0*set_size + idx + offset_b]*2;
    dat_0_vec[1] = dat_0_shared + loc_map[1*set_size + idx + offset_b]*2;
    dat_0_vec[2] = dat_0_shared + loc_map[2*set_size + idx + offset_b]*2;

    mass_centre(dat_0_vec, dat_1 + (idx + offset_b) * 2,
                dat_2 + (idx + offset_b) * 1);

  }
}
```

Figure 6.19: CUDA kernel stub for the `mass_centre` parallel loop. Indirectly-accessed data is staged into shared memory. If the indirectly-accessed data were written to, an additional copy-out after the loop invoking the `mass_centre` function would be generated. This kernel is launched once for each colour.

```
for col in xrange(plan.ncolors):
    # PyCUDA kernel launch
    fun.prepared_async_call(grid_size, block_size,
                            stream, *arglist,
                            shared_size=shared_size)
```

Figure 6.20: The PyOP2 CUDA backend launches a generated kernel once for each partition colour.

device.

Device. This is the same as *Host*, except that the version on the device is considered the currently-correct version of the data.

Both. Space on the host and the device contains the same values. The values on either of these may be used.

Data transitions between these states occur when certain operations are performed. When a state transition occurs, any necessary transfer of data between the host and the device is performed by PyOP2. The user has no involvement in the management of these states or the data transfers, but it is important to understand when transfers occur. Code written without regard for when transfers occur may perform a large number of transfers which would inhibit performance.

Figure 6.21 gives an overview of the state transitions that occur. When a `Dat` is first declared, it is always in the Device Unallocated state. Memory is always allocated on the host, either in the form of a NumPy array that was passed to the `Dat` constructor, or that was allocated by the constructor. The `Dat` moves into the Host state when the data is accessed using the `data` accessor, or when a value is assigned to one of its elements in Python. The data can be accessed in a read-only mode by using the `data_ro` property, which will cause it to move into the Both state if it was already in the Device state, since the data is copied back to the host but will not be invalidated on the device. It moves into the Device state if it is written to by a parallel loop. If a `Dat` is in the Host state when it is read by a parallel loop, it will go into the Both state since the it will be transferred to the device but its values will not be altered.

The compilation, linking, and launching of generated kernels is handled by the PyCUDA and

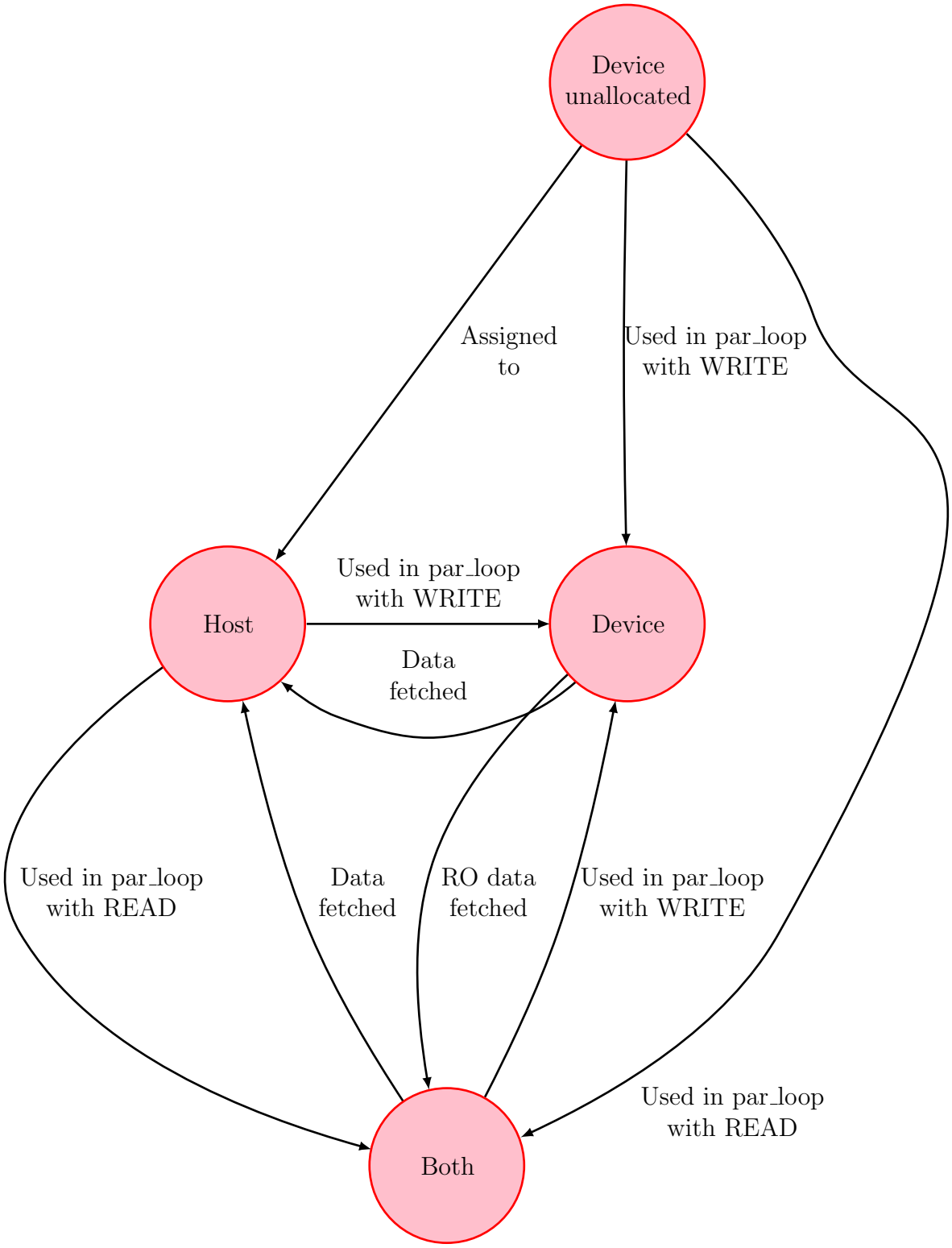


Figure 6.21: Transitions between data states in device backends.

PyOpenCL libraries. These libraries provide a Pythonic interface to access devices capable of executing CUDA and OpenCL code. Both of these libraries also provide a NumPy-like interface to create and modify arrays on the device. This is used to allocate and manage data that underlies the `Dat` objects.

The CUDA backend currently supports all PyOP2 API functions. Linear algebra is handled using the CUSP [BG12] library. Local matrices are all assembled in parallel, and are stored in a temporary array. This temporary array is then assembled into the CSR data structure after the execution of the parallel loop over all partition colours completes. The assembled CSR structure can be passed directly to CUSP when a solve operation begins.

An OpenCL backend has complete support for the PyOP2 API, but at present there is no OpenCL linear algebra library that supports matrix assembly as well as solving. This is worked around by implementing code for assembling CSR matrices in OpenCL within PyOP2, and using PETSc for solving systems. This approach is inconvenient for non-CPU OpenCL targets since it requires the transfer of the assembled matrix from the device back to the host, and the transfer of the solution back to the device after the solve.

The use of ViennaCL [WRS12] for solving an assembled system would also be possible. However, its interface presently accepts pointers to matrix data on the host, and making use of it would require its interface to be changed to accept pointers to data on the compute device. Whilst this is possible, it was not done as part of the work presented in this thesis due to time constraints.

6.5.5 MPI Support

The MPI support in PyOP2 has primarily been authored by Lawrence Mitchell and David Ham. However, we describe MPI support in this thesis to provide a complete outline of PyOP2, and because PyOP2 MPI support is used in the experiments described in the following chapter. We also draw attention to the fact that the MPI implementation strategy of PyOP2 is heavily based on the ideas used in the original OP2 library. The exposition of the MPI support in this section is derived from [Mit13].

```
start_halo_exchanges()
for e in entities:
    if can_assemble(e):
        compute(e)
finish_halo_exchanges()
for e in entities:
    if still_needs_computing(e):
        compute(e)
```

Figure 6.22: Communication and computation overlap strategy in PyOP2. Elements that can be computed on without exchange are assembled whilst exchange takes place. The remaining elements are computed over after the exchange finishes.

```
start_halo_exchanges()
for e in core_entities:
    compute(e)
finish_halo_exchanges()
for e in additional_entities:
    compute(e)
```

Figure 6.23: Execution over core entities takes place whilst halo exchange proceeds. No test is required to check whether an entity can be computed in this strategy.

MPI parallelism is implemented in PyOP2 by distributing the mesh between MPI processes and exchanging halo data when necessary. In order to reduce the time spent idle waiting for data transfers, an attempt is made to overlap communication and computation. The majority of set elements that are held by each node can be processed in parallel without communication. These are the elements that use data which is held by the current process. When a parallel loop is executed, exchange of halo data begins, and then execution over these set elements proceeds. Figure 6.22 presents an overview of the ordering of computation and halo exchanges. Once all the elements that can be visited have been visited, the halo exchange is completed and execution over the remaining elements can begin.

In order to make the computation and halo exchange process as simple and efficient as possible, the elements that can be iterated over whilst halo exchange is in progress are arranged in a contiguous chunk. The set entities that require a halo exchange are placed into another contiguous chunk. This allows transformation of the strategy outlined in 6.22 from one that iterates over all elements twice, to one that iterates over the entities that can be executed over immediately (the *core* entities) then iterates over the other entities. This transformed process is outlined in Figure 6.23.

In the implementation, the entities of a set are divided into the following four groups:

Core entities. These entities are owned by the process and can be processed without halo data.

Owned entities. These entities are owned, but they touch the halo, so they need halo data before they can be processed.

Exec halo. These entities are not owned, but are redundantly executed over because they touch local entities.

Non-exec halo. These entities are not owned and not processed in the parallel loop, but are needed to process the exec halo entities.

The entities in a set are ordered in memory such that all the core entities are first, followed by the owned entities, and then the exec halo, and finally the non-exec halo. This ordering makes it easy to launch separate kernels for core and additional entities on GPUs. Launching separate kernels also avoids branching in the kernel to decide whether to execute on a given entity. The execution over entities requiring the halo exchange is deferred as long as possible.

PyOP2 does not provide a mechanism for partitioning the data into these separate sets, but this could be implemented by following the design of the OP2 partitioner, which makes use of ParMetis or PT-Scotch for computing the partitioning. The experimental work in the following chapter makes use of data that is owned by Fluidity, which computes the partitioning. For this reason, it has not been necessary to implement a PyOP2-based partitioning so far.

6.6 Discussion of Further Development

The design of PyOP2 provides a great deal of flexibility in implementing an unstructured mesh application, since the API embodies few assumptions about Sets and Maps. It is imaginable that further improvements could be made by exploiting higher-level semantic information. This

information could be used to simplify the API in order to make it more intuitive, as well as opening up opportunities for making more radical code transformations. In this section we briefly consider these ideas.

6.6.1 Simplifying Parallel Loops

The parallel loop API presently requires each of the Dat arguments to be specified with a Map as part of their access description. From the programmer's point of view, it is often obvious which Map is suitable for a given iteration set and data set. Furthermore, it is common in a finite element implementation for there to only be one possible Map that can be used in a given context. For example, there will usually be a single mapping from the set of cells to the degrees of freedom for a particular basis.

This property of finite element applications could be exploited to simplify the parallel loop API. When the Map for a given argument can be determined automatically by its uniqueness, the specification of it as part of the argument may be omitted entirely.

We consider an example to demonstrate the simplification of the parallel loop specification. The first parallel loop shown in Figure 6.24 invokes the `mass_centre` kernel presented in the example earlier. The `cell_vertex` Map is the only Map that was defined from cells to vertices, so is the only possible choice for the `vertex_coords` argument. Since the other two arguments are defined on the same set as the iteration set, the identity mapping is the clear choice. The second parallel loop invocation shown in the figure passes the same arguments with the mappings omitted. The parallel loop implementation could select the appropriate Maps for the arguments automatically.

6.6.2 Delayed Evaluation and Kernel Fusion

The present PyOP2 design and implementation requires immediate evaluation of parallel loops at the point at which the `par_loop` function is called. Allowing the evaluation of parallel loops to be delayed will present additional opportunities for optimising the generated code.

```
op2.par_loop(mass_centre, cells,
             vertex_coords(cell_vertex, op2.READ),
             cell_centre(op2.IdentityMap, op2.WRITE),
             cell_mass(op2.IdentityMap, op2.WRITE))

op2.par_loop(mass_centre, cells,
             vertex_coords(op2.READ),
             cell_centre(op2.WRITE),
             cell_mass(op2.WRITE))
```

Figure 6.24: A parallel loop call using the present PyOP2 API, and one using an API specialised to mesh data types.

One opportunity involves the fusion of multiple kernels into a single large kernel. This opens up opportunities for reducing the amount of memory bandwidth used in total by all the kernels. For example, the fusion of kernels that assemble a matrix and a vector over the same mesh (or two matrix assembly kernels over the same mesh) could reduce the amount of data that is read, since some input data will be shared between the kernels.

It is imaginable that array contraction could also reduce data transfer when the output of one kernel is the input to another kernel. However, because PyOP2 code generation is performed at run-time, it is not possible to perform a liveness analysis to determine which arrays can be contracted, since the contracted array may be required later. In order for array contraction to be safely performed, some additional liveness information may need to be specified by the user. Alternatively, arrays could be contracted speculatively, perhaps using information about liveness from a previous execution that was used to gather profiling information. Should the value of a contracted array later be required, its value would have to be computed just-in-time. This would require keeping a trace of previously-executed kernels, so that values that are needed later can be computed.

The implementation of delayed evaluation will require a mechanism to eventually force the evaluation of parallel loop calls. Assuming that a trace of parallel loop calls is recorded, it may become necessary to force evaluation when a trace becomes too long, since otherwise too much memory may be occupied. Alternatively, if the outputs of a trace are required by the user for computations outside of PyOP2, they will be obtained by using the `data_ro` property of a Dat, which must force evaluation in order to return the correct result.

6.7 Conclusion

We have seen that the PyOP2 abstraction is an implementation of the OP2 paradigm, with modifications that make it a suitable parallel execution layer in a finite element code generation toolchain. It improves upon the original OP2 implementation for this purpose in particular by generating code at runtime and providing mechanisms for linear algebra operations. Implementing PyOP2 in Python will enable integration with existing FEniCS toolchain components, and has allowed simplification of the API compared to OP2.

The present PyOP2 design may serve as a foundation for the exploration of further optimisations and implementation of higher-level mesh-based interfaces in future work. The implementation of these optimisations and implementations will further demonstrate the versatility of PyOP2 as a platform for developing mesh-based applications, and as a performance-portable parallel execution layer.

Chapter 7

Firedrake: A Performance-Portable Finite Element Toolchain

7.1 Introduction

In this chapter we present Firedrake, a performance-portable toolchain for finite element implementations that integrates UFL, the FEniCS Form Compiler, Fluidity, and PyOP2. This toolchain provides a fully automated compilation pathway from UFL to multi- and many-core platforms. The Fluidity integration provides a means to implement performance-portable solvers within its existing legacy codebase without requiring an entire complex implementation to be rebuilt from scratch. We will demonstrate that the generated code exceeds the performance of the best available alternatives, without requiring manual tuning or modification of the generated code.

We present performance results for an advection-diffusion problem across C, CUDA, OpenMP, and MPI backends. We show that its performance exceeds that of the best available alternatives in Fluidity and DOLFIN both when comparing sequential and parallel implementations. The same piece of code generated by the form compiler is transformed by PyOP2 to different implementations for the C, CUDA, OpenMP, and MPI backends. This demonstrates that the concerns of the form compiler (FFC) and the runtime system (PyOP2) have been fully

separated in the toolchain.

The experimental work presented in this chapter is presented in [MRM+13]. This publication provides an overview of the Firedrake toolchain, and argues that it provides performance-portability for finite element methods.

We will continue this chapter with an overview of the toolchain, then describe the changes that have been made to FFC and Fluidity. Next, we will present experimental results and and draw conclusions.

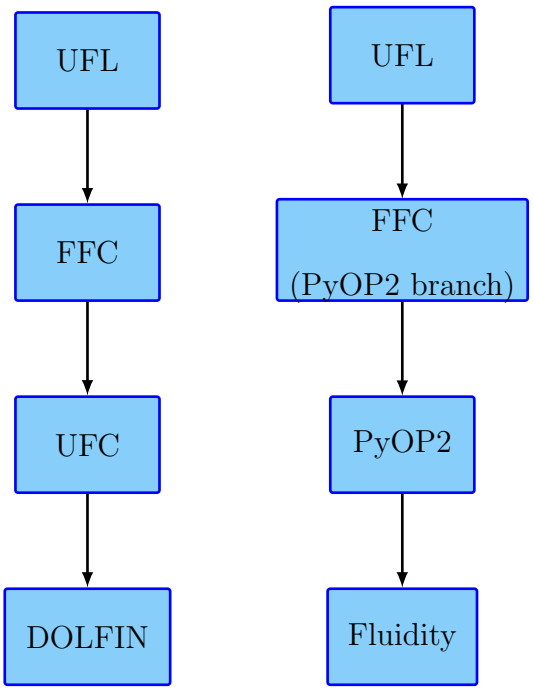
7.2 Toolchain Overview

The Firedrake toolchain is shown in Figure 7.1 alongside the DOLFIN toolchain for comparison. The local assembly kernels produced by FFC are translated into target-specific implementations by PyOP2. Both toolchains use the main branch of the UFL library. The Firedrake toolchain uses a modified version of FFC that can generate PyOP2 kernels instead of UFC code. This modified version of FFC retains compatibility with DOLFIN, and a long-term goal is to merge these additions upstream into FFC.

The Firedrake version of Fluidity has an interface that wraps objects in its Python interface with PyOP2 data structures. This enables users to write finite element methods in UFL in an options file. At runtime, Fluidity passes these forms to PyOP2, which invokes FFC to compile them into kernels that it can execute.

7.2.1 Fluidity Integration

A small set of non-invasive modifications were made to the Fluidity source to incorporate it into the Firedrake toolchain. At the core of Fluidity is a timestepping loop that iterates over each of the fields defined in the simulation, looks up the type of the equation that is used to compute the field values, and dispatches the appropriate routine for solving that equation. Once the



(a) FEniCS/DOLFIN (b) Firedrake

Figure 7.1: Form compilation toolchains.

new values have been computed for all fields, an adaptive timestepping routine increments the simulation time. We note that there are other components in Fluidity simulations such as mesh adaptivity, a Lagrangian detector implementation, and other non-finite element components, but we omit discussion of these for brevity. Their operation is not affected by the modifications we introduce for the the Firedrake toolchain.

In order to support user-defined finite element methods, a new equation type is added to Fluidity. This is achieved by modifying the code that dispatches the correct equation handler to handle an additional case. The new handler invokes Fluidity's Python interface and executes a small amount of Python code to set up the Firedrake environment, before executing the user's code. An overview of the control flow in the modified Fluidity is given in Figure 7.2.

Setting up the Firedrake environment involves augmenting Fluidity's Python state with PyOP2 objects. The PyOP2 objects wrap the Numpy arrays provided by the state interface. The fields that are returned by the Fluidity `state` object are modified so that they can be treated either as Fluidity fields, or as UFL `Coefficient` objects. This enables the use of Fluidity field objects in UFL forms. A convenience function is also provided for constructing `TestFunction`

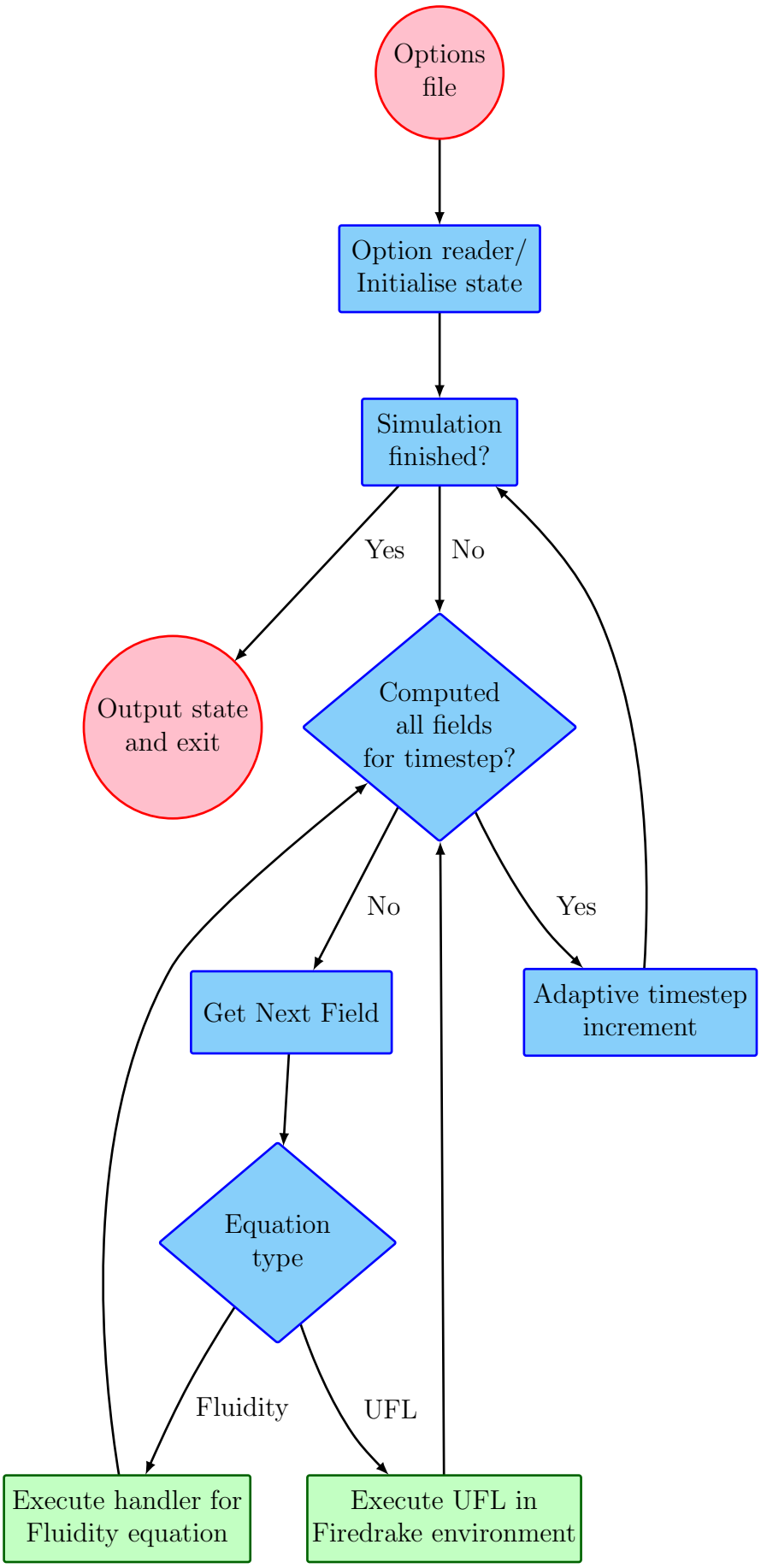


Figure 7.2: Control flow of Fluidity in the Firedrake toolchain.

```
t = state.scalar_fields["Tracer"]

v = TestFunction(t)
u = TrialFunction(t)

a = v*u*dx
L = v*t*dx

# Assemble and solve
solve(a == L, t)
```

Figure 7.3: The complete code required to perform extraction of a tracer field from the Fluidity state in Firedrake. This code would be embedded in a Fluidity option file. The tracer field is used for constructing a `Testfunction` and `TrialFunction`, and its values are used in a form. Passing the field to the solve function causes the result to be stored back into it.

and `TrialFunction` objects from Fluidity fields.

The Firedrake environment also defines a solve function, that uses a similar interface to the DOLFIN solve function. An equation is passed as the first parameter to this function, and the field that the result is returned to is passed as the second parameter. An example of the finite element implementation of the identity equation using the state interface is given in Figure 7.3. Unlike in MCFC, the result from a `solve` does not need to be assigned back to the Fluidity `state` object. This is because the `solve` function is able to store the solution values directly in the array for the field. In the example, `Tracer` field values are updated since the parameter `t` is holding a reference to the `Tracer` field array.

7.3 Modifying FFC for Firedrake

An interface has been implemented in PyOP2 for calling FFC so that UFL form objects can be be compiled into PyOP2 kernels. The interface to this functionality is a single function, `compile_form(form, name)`. A form may contain multiple integrals, which may integrate over different measures. Since integration over different measures requires iteration over different sets, `compile_form` returns a list of kernels over each of the measures.

This interface between PyOP2 and FFC has no dependence on Fluidity, and can be used independently or integrated into a different toolchain. If PyOP2 and FFC are used independently

```
class <name>_<count>_<domain>: public ufc::cell_integral
{
  virtual void tabulate_tensor(double* A,
                               const double * const * w,
                               const ufc::cell& c) const
  {
    <code>
  }
}
```

Figure 7.4: Local tensor evaluation signature in UFC. The array of arrays `w` is used to pass in the values of coefficients. The signature for `tabulate_tensor` is always the same, and only the class name changes.

of Firedrake, it is the responsibility of the user to invoke parallel loops over the correct iteration set with each compiled kernel. In the Firedrake toolchain, the `solve` function invokes form compilation and creates parallel loop calls in order to assemble the matrix and vector that will form a linear system.

Most of the code FFC generates in order to conform to the UFC specification is not required in the Firedrake toolchain. Only the computation of entries of the local tensor is required, since the rest of the required functionality is provided by Fluidity. In the next subsection, we examine the changes that are made to the generation of the local tensor tabulation function.

Function signature

The signature for functions that tabulate the entries of the local tensor in conformance with the UFC specification is shown in Figure 7.4. A pointer to the beginning of the memory holding all the entries of the local tensor is passed in through `A`. Coefficient values are passed in through a double pointer, `w`. The cell upon which the local tensor is to be evaluated is also passed in through `c`, and this provides the vertex coordinates. Because coefficient values are passed in through a double pointer, the signature of the `tabulate_tensor` function is always the same regardless of how many are passed in. Only the class name changes between different tensor tabulation functions, which allows the finite element assembler to distinguish between tensor tabulation functions.

Modification of the function signature to make it a PyOP2 kernel requires the following points

```
void <name>_<count>_integral_<domain>(double A[<ltsize>][<ltsize>],
                                      double *x[<dim>],
                                      ...)
{
  <code>
}
```

Figure 7.5: Local tensor evaluation signature for PyOP2. The local tensor entry and vertex coordinates are the first two arguments. A pointer to Coefficient data is passed as a separate argument for each coefficient.

to be addressed:

- PyOP2 kernels must be C functions.
- Coefficient values will be represented by PyOP2 `Dat` objects. It is not possible to pass in multiple `Dat` objects through a double pointer.
- The vertex coordinates will also be stored in a `Dat`, so they must be passed through a double pointer instead of a `cell` object.
- PyOP2 kernels compute a single entry of a local tensor, rather than all of its entries.

The specification for tensor tabulation kernels that satisfies these constraints is given in Figure 7.5. The class has been removed entirely, and the function name changes to distinguish one tensor tabulation kernel from another. A pointer to a single entry of the local matrix is passed through `A`, which is a two-dimensional array, or in the case where the local tensor is a vector, `A` is a one-dimensional array.

The size of an individual tensor entry is specified in `ltsize`. For example, a tensor with scalar entries has an `ltsize` of 1. For a tensor that has a 2D vector field as its basis, `ltsize` is 2. The coordinates are passed in through the pointers to arrays `x`, and subsequent arguments are pointers to coefficient values. The number of parameters therefore depends on the number of coefficients. The order in which these parameters are passed matches the ID of the coefficients, which are stored in UFL form metadata, and are always numbered from 0.

```
(loop over quadrature points)
  (loop over i)
    (loop over j)
```

Figure 7.6: Loop nest structure inside a `tabulate_tensor` function conforming to the UFC specification. `i` and `j` are loops over the entries of the local tensor

```
(loop over quadrature points)
  (loop over x)
    (loop over y)
```

Figure 7.7: Loop nest structure inside a `tabulate_tensor` function conforming to the PyOP2 specification. `x` and `y` are loops over the dimensions of an individual tensor entry.

7.3.1 Loops Over the Local Tensor

Since PyOP2 kernels only compute a single entry of the local tensor, whereas in UFC the entire local tensor is computed, the structure of loops inside tensor tabulation functions must be changed. Figure 7.6 shows the structure of the loop nest in a kernel that conforms to the UFC specification. In the PyOP2 kernel, the structure of which is shown in Figure 7.7, the loops over the local tensor are removed, and instead loops over the dimensions of an individual entry of the local tensor are inserted.

One downside to this change is that the tensor evaluation kernel now contains code that computes values that are constant for an entire local tensor, but is repeatedly executed because the kernel is called once for each entry of the local tensor. For example, the computation of the Jacobian and its determinant will be repeated even though it does not usually change between tensor entries. A possible extension to the PyOP2 iteration space specification that could mitigate this problem would allow the specification of portions of code that are constant across a local tensor. This issue is likely to be a problem when local tensors are large. Since the experimental work presented in this chapter uses relatively small local matrices, this extension is left for future work.

```
t=state.scalar_fields["Tracer"]
u=state.vector_fields["Velocity"]

p=TrialFunction(t)
q=TestFunction(t)

M=p*q*dx
D=M-0.5*d

adv_rhs = (q*t+dt*dot(grad(q),u)*t)*dx
d=-dt*diffusivity*dot(grad(q),grad(p))*dx
diff_rhs=action(M+0.5*d,t)

solve(M == adv_rhs, t)
solve(D == diff_rhs, t)
```

Figure 7.8: Advection-diffusion UFL implementation in the Firedrake toolchain.

7.4 Experiments

To demonstrate the performance-portability of the Firedrake toolchain, a single-source implementation of a finite element solver for an advection-diffusion equation has been benchmarked on multi- and many-core platforms and compared against a DOLFIN implementation of the same problem. The code that is used to generate the advection-diffusion solver is given in Figure 7.8.

The toolchain-generated code is tested with the sequential, CUDA, MPI and OpenMP backends. Although it would be interesting to benchmark in hybrid MPI/OpenMP mode with one MPI process per socket, hybrid mode execution was not supported by the PyOP2 version used in these experiments. We also benchmark DOLFIN for the same problem, as it is known to generate performant CPU codes and can run in parallel using MPI, and Fluidity's built-in advection-diffusion solver which also supports MPI parallelism.

7.4.1 Experimental Setup

Experiments were performed on a single node of CX1, one of Imperial College's HPC systems. The CX1 node has 2 6-core Intel Xeon X5650 (Westmere-EP, 2.66GHz with 12MB of L3 cache) CPUs, with 16GB of RAM and four Tesla M2050 GPUs. Although multiple GPUs are present in

CX1 nodes, the PyOP2 CUDA backend presently supports single-GPU operation. The CUDA toolkit version 5.0.35 was used to compile code generated by the PyOP2 CUDA backend, and the Intel compiler version 11.1.073 was used for all CPU code. The BLAS and LAPACK used were from Intel MKL 11.1.073 from the Intel suite. The OpenMP implementation was also version 11.1.073 included with the compiler suite. The MPI implementation was provided by Intel MPI version 3.1.038. PETSc 3.3 was used for linear algebra with Fluidity, DOLFIN, and PyOP2 the CPU backends. CUSP 0.3.1 was used for linear algebra with the CUDA backend. All meshes are fully unstructured and are reordered prior to execution using a Hilbert Space-Filling Curve (HSFC) numbering to improve locality. Fluidity provides an interface to the Zoltan HSFC implementation [BDH+07], which was used for this purpose. The unordered meshes were generated using Gmsh [GR09] version 2.5.0.

Execution was launched using the PBS system on CX1 for resource allocation. The configuration line in the PBS script reads:

```
select=1:ncpus=12:mem=4gb:place=excl
```

This selects 12 CPUs and 4GB of memory, and ensures that the job is the only one running on the node. This does not enforce any MPI pinning. OpenMP thread pinning was enforced in the PBS script with:

```
export GOMP_CPU_AFFINITY=0-11
```

Experimental runs were repeated five times and averages are reported, where any outliers are discarded. At the beginning of the experiment, a single timestep is executed in order to force the necessary code generation and any I/O for loading input data and any dynamically-loaded executable code. After this timestep has executed, timing begins, and a further 100 timesteps are executed. After the execution of the last timestep, timing ceases and the recorded time is reported.

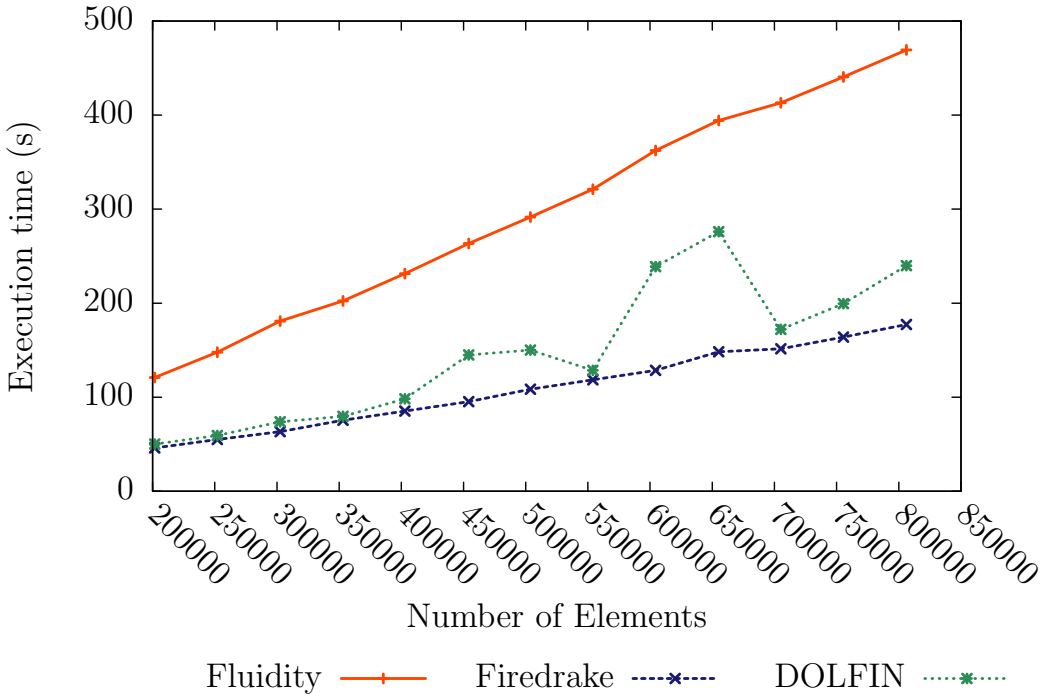


Figure 7.9: Execution times for sequential implementations. The Firedrake implementation outperforms Fluidity and DOLFIN.

7.4.2 Results

Execution times for the sequential implementations are presented in Figure 7.9. We see that the Firedrake implementation outperforms both Fluidity and DOLFIN. The increased execution time of the DOLFIN implementation over Firedrake may be caused by the overhead of the UFC interface. Whilst the Firedrake implementation can perform assembly in a very tight loop with no function calls, the DOLFIN implementation must make calls through the UFC interface.

The Fluidity sequential implementation appears particularly slow compared to Firedrake and DOLFIN. The performance of Fluidity could be optimised, but doing so would reduce its maintainability since it is written in Fortran 90. However, it is clear that the integration of the Firedrake toolchain into Fluidity has enabled a significant speedup without requiring any significant changes to its code. The execution time of the DOLFIN implementation varies depending on the mesh and does not strictly increase with the mesh size. This may be due to a sensitivity to the size or topology of the mesh resulting in different cache behaviour across meshes. However, we note that even in the best cases for DOLFIN, the performance of the

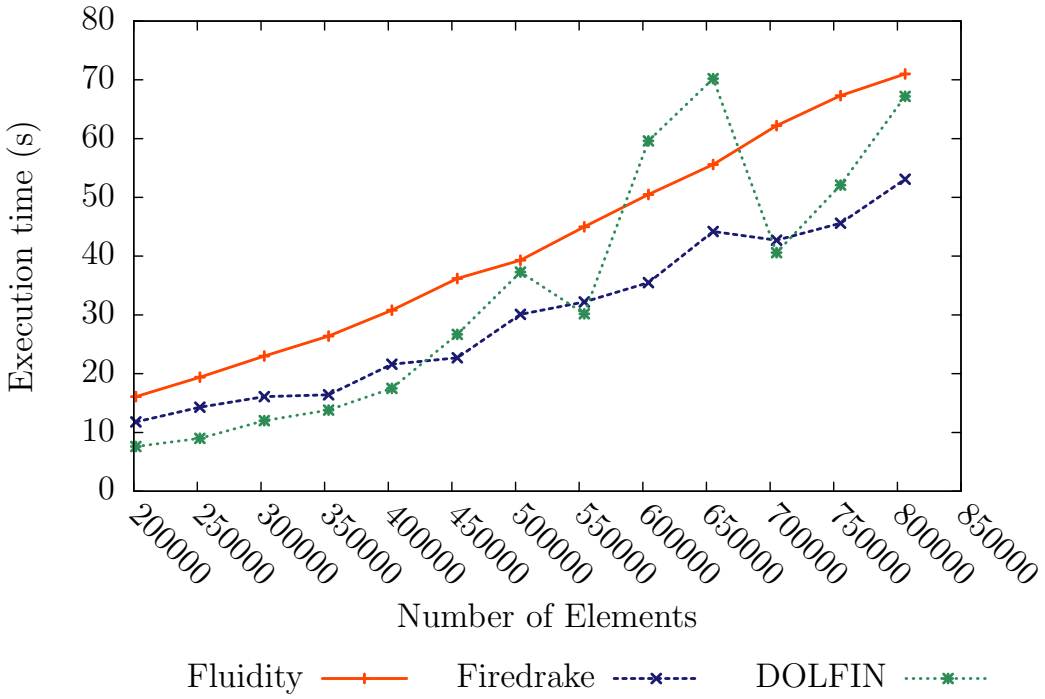


Figure 7.10: Execution times for the MPI implementations.

Firedrake implementation is very similar.

The execution times of MPI implementations are compared in Figure 7.10. The Firedrake implementation generally performs similarly to the DOLFIN implementation. The erratic scaling of DOLFIN seen in the sequential implementation is also present in the MPI implementation. DOLFIN is generally faster on the smaller meshes, but its runtime increases more quickly than Firedrake. The overhead of the Firedrake MPI implementation appears to be slightly higher than that of DOLFIN, but this cost is amortised at around 450,000 elements. For the larger meshes, Firedrake's performance is generally better than that of DOLFIN, with the exception of two meshes where DOLFIN is faster by a small amount.

The performance of different PyOP2 backends in the Firedrake implementation is compared in Figure 7.11. It would be expected that for well-optimised implementations, the OpenMP and MPI backends should have similar performance on the same machine. We see that the OpenMP implementation is slower than the MPI implementation. This could be the result of an inefficient partitioning, which would cause threads to make a large number of accesses to memory locations in a different NUMA domain in the OpenMP implementation. The HSFC

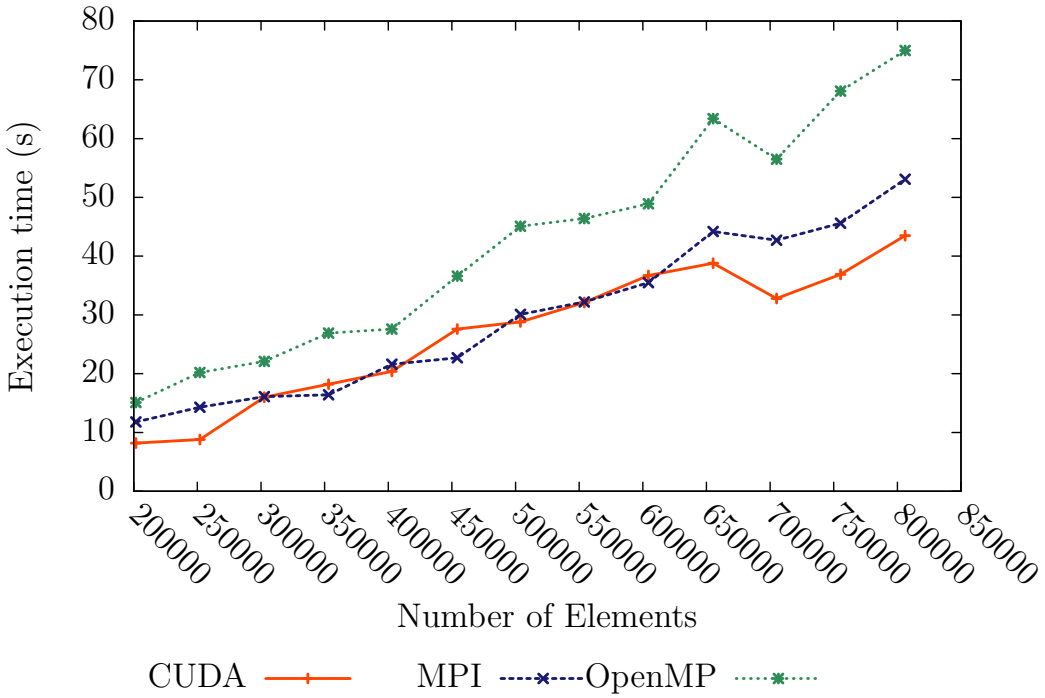


Figure 7.11: Execution times for the Firedrake implementation for different PyOP2 backends.

ordering helps to make contiguous partitions share elements, but this can probably be improved upon further.

The performance of the MPI and CUDA implementations for these hardware platforms is comparable. Because the local matrices are small, this problem is bandwidth bound. Although the peak memory bandwidth of the GPU is double that of the CPUs (144GB/s vs. 2×32GB/s), it is plausible that some level of uncoalesced memory access is occurring due to the unstructured nature of the problem, which will reduce memory bandwidth utilisation. Although we note that there is in general no loss of performance when moving from the MPI backend to the CUDA backend, further investigation is required to determine if the CUDA implementation is making efficient use of the resources available on the GPU, or whether it can be further optimised.

The speedup of the parallel implementations relative to the sequential Fluidity implementation is given in Figure 7.12. The Fluidity sequential run is chosen as a baseline since it is the slowest implementation. The Firedrake/OpenMP times are omitted from this comparison since it requires further optimisation, and the Firedrake/MPI backend executes on the same hardware configuration. It can be seen that the PyOP2 generated code compares favourably

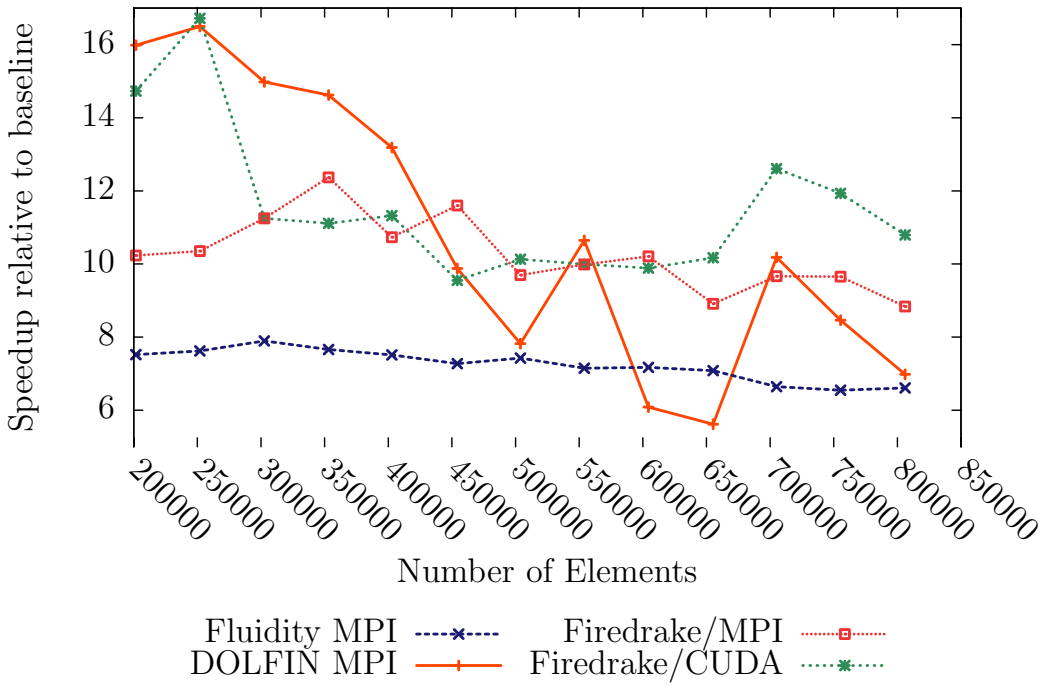


Figure 7.12: Speedup of the parallel implementations relative to the Fluidity sequential implementation, which is chosen as a baseline because it is the slowest. Note the false bottom on this graph.

with the DOLFIN and Fluidity implementations running on 12 cores. The MPI and CUDA implementations exceed the performance of DOLFIN and Fluidity for the largest mesh sizes. The simulation speed for all implementations is constrained by the available bandwidth rather than computational throughput of the target machines.

7.5 Conclusions

The Firedrake toolchain that we have presented in this chapter allows the generation of performance-portable finite element solvers from a single UFL source. Our experimental results demonstrate that Firedrake enables performance-portability, since good performance is obtained on CPU platforms, and similar performance is obtained on the GPU platform, even when both implementations are generated from a single source code. The Firedrake toolchain demonstrates that PyOP2 can be integrated into legacy code without requiring significant changes to that code, whilst enabling performance portability. The combination of performance-portability and mod-

ularity of the toolchain demonstrates that PyOP2 is the right abstraction for building robust, maintainable, and performance-portable finite element solvers.

7.5.1 Further Work

The OpenMP backend provides a speedup over the sequential backend, but further work is required to optimise its performance to match the MPI backend. Alternatively using the OpenMP/MPI backends in a hybrid configuration where one MPI process per NUMA domain is used may alleviate the performance problems of the OpenMP backend. The performance of this configuration should be investigated once support for hybrid mode is added to PyOP2.

Further analysis of the CUDA code could expose whether there is room for more optimisation of the code generated by the CUDA backend. This could be done using a combination of performance modelling, and experiments, which would involve the application of specific code transformations. These methods are required to assess the performance of the CUDA backend since there are no comparable toolchains for generating CUDA code from UFL.

Chapter 8

Conclusion

In this chapter we examine the achievements of this thesis and how they support the claims that are made within. We will discuss and draw attention to the novel and important aspects of the investigations that we have presented. Finally, we will discuss further work and how it should proceed.

8.1 Summary of Thesis Achievements

We review our contributions from Section 1.4.

- *We demonstrate that a single source written in a low-level language cannot be performance portable.* In Chapter 4, we explored the changes in data format and global assembly algorithm that are required to get the best performance on different multi- and many-core architectures, which demonstrated that the optimal choice varies both with the target architecture and problem parameters. These results demonstrated that multiple finite element assembly implementations are required for performance portability.

- *We present the design and implementation of a form compiler that generates code for many-core platforms.* In Chapter 5 we presented the Manycore Form Compiler. The conclusions drawn from the implementation of MCFC provide a guide for the implementation

of a performance-portable parallel execution layer for a finite element form compilation toolchain.

- *We present algorithms that minimise the size of local assembly kernels.* The expression partitioning and placement algorithms discussed in Chapter 5 minimise the code size of local assembly kernels by enabling the generation of loop nests containing small subexpressions. These algorithms are implemented in MCFC.

- *We extend the OP2 abstraction to support linear algebra and runtime code generation in order to use it as a performance-portable parallel execution layer for finite element computations.* Chapter 6 outlines the design of the PyOP2 framework. This extension of the OP2 abstraction provides a suitable parallel execution layer for a performance-portable finite element framework. The addition of linear algebra operations and finite element assembly is supported by the addition of iteration spaces, which add a structured component to the iteration over unstructured meshes.

- *We use PyOP2 as part of a framework for building maintainable, robust, performance-portable finite element solvers.* In Chapter 7 we present Firedrake, which incorporates UFL, FFC, PyOP2, and Fluidity to provide a performance-portable finite element toolchain. The minimal changes required to Fluidity and FFC to accommodate PyOP2's requirements demonstrate that it is a suitable abstraction for finite element assembly. The performance-portability of a single-source implementation in the Firedrake toolchain was demonstrated by benchmarking an advection-diffusion test problem that obtained good performance on multi- and many-core architectures.

8.2 Discussion

The PyOP2 abstraction differs from UFC in the finite element context by providing more control to the runtime about how finite element assembly is executed, but provides a less structured interface for finite element assembly. As a result, there is more flexibility in how a finite element assembler can be implemented with PyOP2 but more work is required to define

and manage suitable mesh abstractions based on PyOP2 data structures. One possibility for resolving this issue involves defining further finite-element specific abstractions for meshes and function spaces on top of PyOP2.

A key insight of the design of PyOP2 is the addition of iteration space API, which combines execution with structured iterations, as in Æcute, with execution over unstructured iterations as in OP2. This idea enables PyOP2 to have the freedom to implement the traversal of both the structured and unstructured iteration spaces, but the separation imposed by the PyOP2 kernel API may restrict the possibility of efficiently implementing assembly of large local matrices. As discussed in Section 7.3.1, the kernel API forces repeated evaluation for each local matrix entry of the values that are constant over an element, such as the Jacobian. The present implementation also allows mixing of function spaces on the same basis by specifying the dimension of a `Dat` to represent vector function spaces, but does not easily enable mixed spaces of different basis functions.

There is presently a strong separation between the structured and unstructured components of the iteration space. The iteration space API can only be used to assemble matrices and vectors. Further generalisation of the notion of an iteration space would allow structured components of `Set` objects, in order to allow iteration over data on structured portions in a more natural fashion.

It is debatable whether the PyOP2 abstraction in its present form provides sufficient generality for a wide range of finite element methods. In particular, it will be important to find ways to enable the use of mixed function spaces whilst retaining performance portability. Another important area of investigation will be to ensure that computations that are invariant across multiple kernel invocations can be moved outside of the loop that the kernel is executed in. The iteration space API may be extended to support this idea of a semi-structured iteration space.

We consider how these investigations should progress in the following section.

8.3 Future Work

The design of iteration spaces and the interface to FFC allows for the construction of solvers for scalar- and vector-valued functions. However, it does not provide a way to specify the assembly of forms that use mixed function spaces (for example, in a form that couples pressure and velocity). This extension will allow the construction of solvers for a variety of equations that are commonly used in modelling physical phenomena, such as the Shallow-Water Equations, the Navier-Stokes Equations and various hyperelasticity equations.

The mesh abstractions provided by PyOP2 are presently very primitive. Whilst this provides flexibility when implementing a mesh-based solver, many decisions about the mesh abstractions must be made by the user of PyOP2. The design and implementation of higher-level mesh abstractions could reduce the overhead and number of choices when developing a finite element application, whilst still enabling performance portability.

The expression partitioning and placement algorithms that enable MCFC to generate loop nests containing subexpressions provide a reduction in the generated code size. However, these algorithms have not been incorporated into the Firedrake toolchain. The incorporation of this code generation strategy in FFC may provide gains in efficiency, particularly for complicated forms on many-core architectures, where the constraints on registers are particularly high.

It will be important to allow the movement of code that computes invariant results in local tensor assembly out of inner loops. This loop-invariant code motion will increase the efficiency of execution when using higher-order elements that result in larger local tensors. The gains in efficiency will be achieved by reducing the amount of computation that is required for each set element.

The power of the code generation approach is that it allows exploration of alternative code generation schemes. Intra-kernel vectorisation for the PyOP2 CPU backends, using different algorithms for global assembly, and different memory management strategies for the OpenMP backend are areas of investigation that will enable the generation of code with improved performance. Adaptation of the generated code to the application context and hardware capabilities

will be enabled through the investigation of these optimisations.

Glossary

Access Descriptor Consisting of an Access Specifier and an Iteration Space, an Access Descriptor tells OP2 or PyOP2 how to access the data for an argument to a parallel loop.

Access Specifier In OP2 and PyOP2, an Access Specifier describes whether an argument to a parallel loop is read, written, incremented, read and written without conflict, or reduced into.

Addto The Addto algorithm is used to construct a finite element global matrix from the local matrices and local-to-global mapping.

AST An Abstract Syntax Tree, which provides a structured representation of statements and expressions.

Coalescing Coalesced memory accesses are achieved on GPUs when threads in one half of a warp all access memory within the same 64-byte window. The accesses are bundled into a single transaction, improving performance.

Dat Data associated with a Set in OP2 or PyOP2. A Dat holds an item of data for every Set element.

Data Set The set that a Map in OP2 or PyOP2 maps to.

DOLFIN The Dynamic Object-Oriented Library for Finite Elements, an interactive problem-solving environment.

FFC The FEniCS Form Compiler, that transforms variational forms written in UFL into UFC code.

FIAT The Finite Element Automatic Tabulator, that tabulates the values of basis functions.

Instant A Python library for dynamically compiling C code and linking it back in to a running Python interpreter.

Iteration Set The Set that is iterated over by a parallel loop in PyOP2. This may also refer to the set that a Map maps from.

Iteration Space A set of index values that will be visited. A kernel is executed for each point in the iteration space.

Kernel In OP2 and PyOP2, a kernel is a function that computes an output of a parallel loop for one Iteration Set entity.

LMA The *Local Matrix Approach*, in which a Global Matrix is not assembled, but the local matrices and local-to-global mappings are used to compute a sparse matrix-vector product.

Map A list of integers that provides a mapping from one Set to another in OP2 or PyOP2. The arity of a Map must be the same for every element of the Set that it maps from.

Mini-partition A partition of the Iteration Set in OP2 or PyOP2. Its size is chosen such that all the associated data for the execution of a parallel loop fits within the cache or Shared Memory.

Plan In OP2 and PyOP2, a plan represents an efficient partitioning and execution schedule for a parallel loop.

Set A set of entities in OP2 or PyOP2. Examples include cells, vertices, edges, etc.

Shared Memory On a GPU, the shared-memory is a small (typically 16-48KB) software-controlled cache.

SpMV Sparse Matrix-Vector Product.

UFC Unified Form Assembly Code, a C++ header specification for generated finite element assembly code.

UFL The Unified Form Language, a domain specific language for writing variational forms.

Vector Map An argument to a parallel loop that gathers all of the Data Set elements for a given Iteration Set element in a parallel loop in PyOP2.

Warp A warp is a set of 32 threads that share a program counter during execution on a GPU.

Bibliography

[ABB+06] 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.

[ALØ+13] M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells. Unified Form Language: A domain-specific language for weak formulations of partial differential equations. *ACM Trans Math Software*, 2013. To appear.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[AMD10] AMD. AMD Radeon 5870 Specifications. http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/hd-5870/Pages/ati-radeon-hd-5870-overview.aspx#2, Retrieved 30 Sep, 2010.

[App10] Applied Modelling and Computation Group, Department of Earth Science and Engineering, South Kensington Campus, Imperial College London, London, SW7 2AZ, UK. *Fluidity Manual*, version 4.0-release edition, November 2010. available at http://hdl.handle.net/10044/1/7086.

[BBC+94] 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.

[BBC+02] 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.

[BBG+09] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc Web page, 2009. http://www.mcs.anl.gov/petsc.

[BBL+12] C. Bertolli, A. Betts, N. Loriant, G. R. Mudalige, D. Radford, D. Ham, M. B. Giles, and P. H. J. Kelly. Compiler optimizations for industrial unstructured mesh CFD applications on GPUs. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2012.

[BCG94] D. A. Burgess, P. I. Crumpton, and M. B. Giles. A parallel framework for unstructured grid solvers. In *Proceedings of the Second European Computational Fluid Dynamics Conference*, pages 391–396. John Wiley and Sons, 1994.

[BDH+07] Erik Boman, Karen Devine, Robert Heaphy, Bruce Hendrickson, Vitus Leung, Lee Ann Riesen, Courtenay Vaughan, Umit Catalyurek, Doruk Bozdag, William Mitchell, and James Teresco. *Zoltan 3.0: Parallel Partitioning, Load Balancing, and Data-Management Services; User's Guide*. Sandia National Laboratories, Albuquerque, NM, 2007. Tech. Report SAND2007-4748W http://www.cs.sandia.gov/Zoltan/ug_html/ug.html.

[Bel03] Richard Bellman. *Dynamic Programming*. Dover Publishing, 2003.

[BG12] Nathan Bell and Michael Garland. CUSP: Generic parallel algorithms for sparse matrix and graph computations, 2012. Version 0.3.1.

[BHK07] W. Bangerth, R. Hartmann, and G. Kanschat. *deal.II - A general-purpose object-oriented finite element library*. ACM New York, NY, USA, 2007.

[BP10] T. Brandvik and G. Pullan. SBLOCK: A framework for efficient stencil-based PDE solvers on multi-core platforms. In *IEEE 10th International Conference on Computer and Information Technology (CIT), 2010*, pages 1181–1188, July 2010.

[BS04] Babak Bagheri and L. Ridgway Scott. About Analysa. Technical Report TR-2004-09, University of Chicago, 2004.

[CCLM11] Andrew Corrigan, Fernando Camelli, Rainald Löhner, and Fernando Mut. Semi-automatic porting of a large-scale Fortran CFD code to GPUs. *International Journal for Numerical Methods in Fluids*, 2011.

[CL11] Andrew Corrigan and Rainald Löhner. Semi-automatic porting of a large-scale CFD code to multi-graphics processing unit clusters. *International Journal for Numerical Methods in Fluids*, 2011.

[CLD11] C. Cecka, A.J. Lew, and E. Darve. Assembly of finite element methods on graphics processors. *International Journal for Numerical Methods in Engineering*, 85(5):640–669, 2011.

[CM90] A.J. Chorin and J.E. Marsden. *A Mathematical-introduction to Fluid Mechanics*. Oklahoma Notes. Springer-Verlag, 1990.

[CMP01] Robert D. Cook, David S. Malkus, and Michael E. Plesha. *Concepts and applications of finite element analysis*. Wiley, 4 edition, October 2001.

[CSB11] M. Christen, O. Schenk, and H. Burkhart. PATUS: a code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676–687, May 2011.

[CSKK11a] C. D. Cantwell, S. J. Sherwin, R. M. Kirby, and P. H. J. Kelly. From h to p Efficiently: Selecting the Optimal Spectral/hp Discretisation in Three Dimensions. *Mathematical Modelling of Natural Phenomena*, 6(3):84–96, 2011.

[CSKK11b] C.D. Cantwell, S.J. Sherwin, R.M. Kirby, and P.H.J. Kelly. From h to p efficiently: Strategy selection for operator evaluation on hexahedral and tetrahedral elements. *Computers and Fluids*, 43(1):23–28, 2011.

[Dev09] Advanced Micro Devices. AMD: Unleashing the Power of Parallel Compute. SIGGRAPH Asia presentation, http://sa09.idav.ucdavis.edu/docs/SA09_AMD_IHV.pdf, 2009.

[DG05] P. Dular and C. Geuzaine. *GetDP Reference Manual*, 2005.

[DH11] Zachary DeVito and Pat Hanrahan. Designing the Language Liszt for Building Portable Mesh-based PDE Solvers. In *SciDAC 2011*, Denver, CO, 2011.

[DJP+11] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. Liszt: A domain specific language for building portable mesh-based PDE solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1–9:12, New York, NY, USA, 2011. ACM.

[DPD11] Denys Dutykh, Raphaël Poncet, and Frédéric Dias. The VOLNA code for the numerical modeling of tsunami waves: Generation, propagation and inundation. *European Journal of Mechanics-B/Fluids*, 30(6):598–615, 2011.

[FMFM13] J. Filipovic, M. Madzin, J. Fousek, and L. Matyska. Optimizing CUDA code by kernel fusion – application on BLAS. *Submitted to Parallel Computing (Elsevier)*, May 2013. http://arxiv.org/abs/1305.1183.

[FPF09a] 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.

[FPF09b] 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.

[Gil12a] Mike Giles. OP2 C++ user's manual. http://www.oerc.ox.ac.uk/research/op2/op2-docs/user.pdf. Retrieved Mar 4th, 2013, 2012.

[Gil12b] Mike Giles. OP2 developers' guide. http://www.oerc.ox.ac.uk/research/op2/op2-docs/dev.pdf. Retrieved Mar 4th, 2013, 2012.

[GMS+10] Mike B. Giles, Gihan R. Mudalige, Z. Sharif, G. R. Markall, and P. H. J. Kelly. Performance analysis of the OP2 multi-layer abstraction framework on many-core architectures. In *PMBS10: Performance Modelling, Benchmarking and Simulation of High Performance Computing Systems at SC '10*, November 2010.

[GMS+11] M.B. Giles, G.R. Mudalige, Z. Sharif, G. Markall, and P.H.J. Kelly. Performance analysis and optimization of the OP2 framework on many-core architectures. *The Computer Journal*, 2011.

[Gol89] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, Reading, MA, 1989.

[GR09] Christophe Geuzaine and Jean-François Remacle. Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities. *Int. J. Numerical Methods in Engineering*, 79(11):1309–1331, 2009.

[Hak12] Johan Hake. FFC Bug 956979. https://bugs.launchpad.net/ffc/+bug/956979. Retrieved Mar 4th, 2013, March 2012.

[HFG+09] D. A. Ham, P. E. Farrell, G. J. Gorman, J. R. Maddison, C. R. Wilson, S. C. Kramer, J. Shipton, G. S. Collins, C. J. Cotter, and M. D. Piggott. Spud 1.0: generalising and automating the user interfaces of scientific computer models. *Geoscientific Model Development*, 2(1):33–42, 2009.

[HFK06] Anup Hosangadi, Farzan Fallah, and Ryan Kastner. Optimizing polynomial expressions by algebraic factorization and common subexpression elimination. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(10):2012–2022, 2006.

[HLDK09] 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.

[HLKD09] Lee Howes, Anton Lokhmotov, Paul H.J. Kelly, and Alastair F. Donaldson. Decoupled access/execute metaprogramming for GPU-accelerated systems. In *Symposium on Application Accelerators in High Performance Computing (SAAHPC)*, 2009.

[HNW93] E Hairer, SP Nørsett, and G Wanner. Solving ordinary differential equations I: nonstiff problems. 1993.

[Hof05] H. Peter Hofstee. Power efficient processor architecture and the Cell processor. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 258–262, Washington, DC, USA, 2005. IEEE Computer Society.

[How10] Lee William Howes. *Indexed dependence metadata and its applications in software performance optimisation.* PhD thesis, Imperial College London, 2010.

[HPHO05] F. Hecht, O. Pironneau, A. L. Hyaric, and K. Ohtsuka. *FreeFEM++ Manual*, 2005.

[HW07] Jan S. Hesthaven and Tim Warburton. *Nodal Discontinuous Galerkin Methods.* Springer, 2007.

[IMWe12] Kent-Andre Mardal Ilmar M. Wilbers and Martin S. Alnæes. *Instant: just-in-time compilation of C/C++ in Python*, chapter 14. In Logg et al. [LMW+12], 2012.

[JHJ12] N. Jansson, J. Hoffman, and J. Jansson. Framework for massively parallel adaptive finite element computational fluid dynamics on tetrahedral meshes. *SIAM Journal on Scientific Computing*, 34(1):C24–C41, 2012.

[Joh11] Anders E. Johansen. FFC Bug 897372. https://bugs.launchpad.net/ffc/+bug/897372. Retrieved Mar 4th, 2013, November 2011.

[KCO+10] S. Kamil, Cy Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, April 2010.

[KEGM10] D. Komatitsch, G. Erlebacher, D. Göddeke, and D. Michea. High–order finite–element seismic wave propagation modeling with MPI on a large GPU cluster. *Journal of Computational Physics*, 229:7692–7714, June 2010. DOI 10.1016/j.jcp.2010.06.024.

[Kel95] C.T. Kelley. *Iterative Methods for Linear and Nonlinear Equations*. Frontiers in Applied Mathematics. Society for Industrial and Applied Mathematics, 1995.

[KGEM10] Dimitri Komatitsch, Dominik Göddeke, Gordon Erlebacher, and David Michéa. Modeling the propagation of elastic waves using spectral elements on a cluster of 192 GPUs. *Computer Science Research and Development*, 25(1-2):75–82, 2010.

[KK98] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[KL06] R. C. Kirby and A. Logg. A compiler for variational forms. *ACM Transactions on Mathematical Software*, 32(3):417–444, 2006.

[KL12] Robert C. Kirby and Anders Logg. *Tensor Representation of Finite Element Variational Forms*, chapter 8. In Logg et al. [LMW+12], 2012.

[KLRT12] Robert C. Kirby, Anders Logg, Marie E. Rognes, and Andy R. Terrel. *Common and Unusual Finite Elements*, chapter 3. In Logg et al. [LMW+12], 2012.

[KME09] 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.

[KS99] George E. M. Karniadakis and Spencer J. Sherwin. *Spectral/hp Element Methods for CFD*. Oxford University Press, 1999.

[KT13] Matthew G. Knepley and Andy R. Terrel. Finite element integration on GPUs. *ACM Trans. Math. Softw.*, 39(2):10:1–10:13, February 2013.

[KWBH09] A. Klöckner, T. Warburton, J. Bridge, and J.S. Hesthaven. Nodal discontinuous Galerkin methods on graphics processors. *Journal of Computational Physics*, In Press:–, 2009.

[Lan03] Hans Petter Langtangen. *Computational Partial Differential Equations, Numerical Methods and Diffpack Programming*. Springer-Verlag, 2003.

[LMW+12] Anders Logg, Kent-Andre Mardal, Garth N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012.

[LNOM08] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, 2008.

[Lon03] K.R. Long. Sundance rapid prototyping tool for parallel PDE optimization. *Large-scale PDE-constrained optimization*, page 331, 2003.

[LW10] Anders Logg and Garth N. Wells. DOLFIN: Automated finite element computing. *ACM Trans. Math. Softw.*, 37(2):20:1–20:28, April 2010.

[Mat09] The MathWorks. MATLAB documentation. http://www.mathworks.com/access/helpdesk/help/techdoc/matlab.html, Retrieved 17 Oct 2009, 2009.

[MGS+12] G. R. Mudalige, M. B. Giles, B. Spencer, C. Bertolli, and I. Z. Reguly. Designing OP2 for GPU Architectures. Journal of Parallel and Distributed Computing, In Press. http://www.sciencedirect.com/science/article/pii/S0743731512001694, August 2012.

[MHK10] Graham R. Markall, David A. Ham, and Paul H.J. Kelly. Towards generating optimised finite element solvers for GPUs from high-level specifications. *Procedia Computer Science*, 1(1):1809 – 1817, 2010. ICCS 2010.

[Mit13] Lawrence Mitchell. Partitioning and numbering meshes for efficient MPI-parallel execution in PyOP2. http://fenicsproject.org/pub/workshops/fenics13/slides/Mitchell.pdf Presented to the FEniCS '13 conference, Cambridge, UK, March 2013.

[MK09] 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.

[MRM+13] G. R. Markall, F. Rathegeber, L. Mitchell, N. Loriant, C. Bertolli, D. A. Ham, and P. H. J. Kelly. Performance portable finite element assembly using PyOP2 and FEniCS. In *Proceedings of the International Supercomputing Conference (ISC) '13*, volume 7905 of *Lecture Notes in Computer Science*, June 2013. In press.

[MSH+13] G. R. Markall, A. Slemmer, D. A. Ham, P. H. J. Kelly, C. D. Cantwell, and S. J. Sherwin. Finite element assembly strategies on multi-core and many-core architectures. *International Journal for Numerical Methods in Fluids*, 71(1):80–97, 2013.

[Mun11] Aaftab Munshi. The OpenCL Specification: version 1.2 revision 15. http://developer.amd.com/wordpress/media/2012/10/opencl-1.2.pdf. Retrieved Jun 14 2013, November 2011.

[NVi10a] NVidia. Fermi white paper. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf, Retrieved 19 May 2011., 2010.

[NVi10b] NVidia. NVidia Geforce GTX480 Specifications. http://www.nvidia.com/object/product_geforce_gtx_480_us.html, Retrieved 30 Sep, 2010.

[NVi12a] NVidia. CUDA C Programming Guide Version 5.0. URL: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. Retrieved Jun 14 2013, October 2012.

[NVi12b] NVidia. CUDA Visual Profiler User Guide Version 5.0. URL: http://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf. Retrieved Jun 14 2013, October 2012.

[NVi13] NVidia. NVidia's next-generation CUDA compute architecture: Kepler GK110. URL: http://www.nvidia.co.uk/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf. Retrieved Jun 14, 2013.

[Oli06] Travis E. Oliphant. *Guide to NumPy*. Provo, UT, March 2006.

[Pig10] M. D. Piggott. About ICOM. http://amcg.ese.ic.ac.uk/index.php?title=ICOM, Retrieved 30 Sep, 2010.

[PMJ+05] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[RK13] Francis P. Russell and Paul H. J. Kelly. Optimized code generation for finite element local assembly using symbolic manipulation. *ACM Transactions on Mathematical Software*, 2013. In press.

[RMKB11] Francis P Russell, Michael R Mellor, Paul HJ Kelly, and Olav Beckmann. DESOLA: an active linear algebra library using delayed evaluation and runtime code generation. *Science of Computer Programming*, 76(4):227–242, 2011.

[RMM+12] F. Rathgeber, G. R. Markall, L. Mitchell, N. Loriant, D. A. Ham, C. Bertolli, and P. H. J. Kelly. PyOP2: A High-Level Framework for Performance-Portable Simulations on Unstructured Meshes. In *WOLFHPC2012: Workshop on Languages for High-Performance Computing at SC '12*, November 2012.

[Rus11] Francis Russell. *An Active-Library Based Investigation into the Performance Optimisation of Linear Algebra and the Finite Element Method*. PhD thesis, Imperial College London, June 2011.

[Sal09] Jerome Saltzer. *Principles of computer system design: an introduction*. Morgan Kaufmann, 2009.

[SSD03] Pavel Solin, Karel Segeth, and Ivo Dolezel. *Higher-Order Finite Element Methods*. Chapman and Hall/CRC, 2003.

[TCK+11] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The Pochoir stencil compiler. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM.

[UCB11] Didem Unat, Xing Cai, and Scott B. Baden. Mint: realizing CUDA performance in 3D stencil methods with annotated C. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 214–224, New York, NY, USA, 2011. ACM.

[UZC+12] D. Unat, Jun Zhou, Yifeng Cui, S.B. Baden, and Xing Cai. Accelerating a 3D finite-difference earthquake simulation with a C-to-CUDA translator. *Computing in Science Engineering*, 14(3):48–59, May-June 2012.

[VEB+11] Peter E. J. Vos, Claes Eskilsson, Alessandro Bolis, Sehun Chun, Robert M. Kirby, and Spencer J. Sherwin. A generic framework for time-stepping partial differential equations (PDEs): general linear methods, object-oriented implementation and application to fluid problems. *Int. J. Comput. Fluid Dyn.*, 25(3):107–125, March 2011.

[VG98] Todd L. Veldhuizen and Dennis Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *In Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO98)*. SIAM Press, 1998.

[Vos11] Peter E.J. Vos. *From h to p efficiently: Optimising the implementation of spectral/hp element methods*. PhD thesis, Imperial College London, 2011.

[VSK10] Peter E. J. Vos, Spencer J. Sherwin, and Robert M. Kirby. From h to p efficiently: Implementing finite and spectral/hp element methods to achieve optimal performance for low- and high-order discretisations. *J. Comput. Phys.*, 229(13):5161–5181, July 2010.

[WRS12] J. Weinbub, K. Rupp, and S. Selberherr. Towards Distributed Heterogenous High-Performance Computing with ViennaCL. In *LSSC'11: Proceedings of the 8th International Conference on Large-Scale Scientific Computing*, pages 359–367, Berlin, Heidelberg, 2012. Springer-Verlag.

[ZM12] Yongpeng Zhang and Frank Mueller. Auto-generation and auto-tuning of 3D stencil codes on GPU clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 155–164, New York, NY, USA, 2012. ACM.