

ON CODE GENERATION TECHNIQUES FOR FINITE ELEMENTS

MIKLÓS HOMOLYA

Department of Computing
Imperial College London

June 2018

Submitted in part fulfilment of the requirements
for the degree of Doctor of Philosophy
and the Diploma of the Imperial College

Declaration

I herewith certify that all material in this dissertation which is not my own work has been properly acknowledged.

Miklós Homolya

The copyright of this thesis rests with the author and is made available under a Creative Commons Attribution Non-Commercial No Derivatives licence. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or redistribution, researchers must make clear to others the licence terms of this work.

Abstract

Efficient numerical solvers for partial differential equations are critical to vast fields of engineering and scientific research, including weather and climate prediction. State-of-the-art finite element discretisations, although offering many desirable properties, are often difficult to implement. Consequently, many developers of complicated finite element discretisations use code generation based software platforms. The *form compiler* is a key component of such software: it takes a high-level description of the weak form of partial differential equations and produces low-level code that carries out the finite element assembly.

This thesis presents a novel, multi-stage code generation framework for the finite element method, implemented in the *Two-Stage Form Compiler* (TSFC). A core idea of TSFC is the introduction of a tensor algebra language as its *intermediate representation*. This creates a clean separation between the implementation of finite element objects and geometric terms as well as the problem of efficient code generation for tensor algebra expressions, which enables the maintenance of the structure of the input expression longer in the compiler pipeline than previous form compilers, and thus helps to facilitate the application of optimisations at the highest possible level of abstraction.

The intermediate language is also proven to be a suitable medium and level of abstraction for numerous optimisations, including the expression and exploitation of structure intrinsic to *some* finite elements. This includes sum factorisation on cuboid cells for continuous, discontinuous, $H(\text{div})$ and $H(\text{curl})$ conforming elements. My experiments confirm optimal algorithmic complexity for high-order finite element assembly. This is achieved through several novel contributions: the introduction of a more powerful interface between the form compiler and the library providing the finite elements; a more abstract, smarter library of finite elements that explicitly communicates the structure of elements; and form compiler algorithms to automatically exploit this exposed structure.

To Our Lady, Seat of Wisdom

Acknowledgements

First and foremost, I would like to thank my supervisor David Ham and my co-supervisor Lawrence Mitchell. David leads with high integrity, and I could always rely on him when his help was needed. I am especially grateful that I did not have to fear losing a sense of direction since he has an inventory of ideas on what could be done next. Lawrence has expert knowledge of virtually everything our work has touched upon, from fluids through numerical methods to programming modern hardware. He has given me immense help both in understanding the theory as well as the implementation in preexisting software.

I am also grateful for the kind help and support of senior PhD students in the early years of my PhD: Andrew McRae, Fabio Luporini, and Doru Bercea. I am glad that later I could help junior PhD students myself: Thomas Gibson and Tianjiao Sun. I am happy that they appreciated my kindness during code reviews, and it was a joy to see them grow in the craft of developing production software. I would also like to express my gratitude to my officemates in the William Penney Laboratory for creating a great work environment, especially Francis Russell and Emanuele Vespa. We are all passionate about C++, and we had numerous great conversations. Special thanks to Prof Paul Kelly for his friendly support and diverse connections.

Furthermore, I would like to say thanks to Colin Cotter, whom I saw more often than some of his own supervisees. I am also grateful for the remote collaborations with Jan Blechta, who ported TSFC to FEniCS as an FFC representation, and Robert C. Kirby, whose comments and advice greatly improved my third paper.

My PhD funding was provided courtesy of the Grantham Institute. This work used the ARCHER UK National Supercomputing Service (<http://www.archer.ac.uk>).

Contents

1	Introduction	1
1.1	Thesis statement	2
1.2	Technical contributions	2
1.3	Dissemination	3
1.4	Thesis outline	4
2	Background	7
2.1	Preliminaries	7
2.1.1	From global assembly to local assembly	11
2.1.2	Changing to reference space	13
2.1.3	Numerical integration and assembly code	15
2.1.4	Structure within finite elements	16
2.1.5	Discontinuous Galerkin methods	17
2.2	Previous work on form compilers	20
2.3	High-order finite elements in numerical software	22
2.4	Firedrake overview	23
3	Consistent edge orientations for quadrilateral meshes	27
3.1	Problem statement	29
3.2	Proof of the existence of a solution	29
3.3	Serial algorithm	32
3.4	Parallel algorithm	32
3.5	Experiments	37
3.6	Alternative approaches to parallelism	40
4	Two-Stage Form Compiler (TSFC)	45
4.1	Compilers as series of transformations	46

4.2	On preserving structure	47
4.3	Tensor algebra and index notation in UFL	50
4.4	UFL preprocessing and modified terminals	52
4.5	Substitution of modified terminals	53
4.6	The intermediate representation: GEM	55
4.7	Basic optimisations	57
4.7.1	Constant folding for scalars	57
4.7.2	Cellwise constant quantities	57
4.7.3	Fast Jacobian evaluation for affine cells	58
5	FInAT: exposing structure in finite elements	61
5.1	Generic FIAT element wrapper	62
5.2	Vector and tensor elements	62
5.3	Tensor product element	63
5.4	Collocated quadrature points	63
5.5	Enriched element	65
5.6	Value modifier element wrappers	65
6	Automatically exploiting the structure in finite elements	67
6.1	Related work	67
6.2	Delta cancellation: simple case	69
6.3	Sum factorisation: Laplace operator	71
6.4	Sum factorisation: parametrised forms	76
6.5	Delta cancellation: nonzero patterns	76
6.6	Splitting Concatenate nodes	78
6.7	Order of transformations	81
7	Code generation: GEM $\rightarrow$ C	83
7.1	Elimination of ComponentTensor nodes	83
7.2	Generation of loop nests	84
7.3	Topological ordering of loop nests	91
7.4	Unrolling short IndexSum loops	94
7.5	COFFEE integration	94
8	Experimental evaluation	97
8.1	Test cases	97

8.2	Form compilation speed	100
8.3	Algorithmic complexity for high-order FEM	102
8.3.1	Matrix assembly	103
8.3.2	Operator action and linear forms	106
9	Summary & outlook	113
9.1	Summary	113
9.2	Future work	115
9.2.1	Sum factorisation on simplicial cells	115
9.2.2	Low-level performance on modern hardware	115
9.2.3	Transferring remaining COFFEE functionality	116
9.2.4	Global assembly of sparse local tensors	116
9.2.5	Countering beta expansion	117
9.2.6	Sum factorised evaluation of <i>future</i> facet integrals	118
9.2.7	Finite elements with some continuity of derivatives	119
9.2.8	Dual evaluation in FInAT	120
9.2.9	Generating GEM from SLATE?	120
	Bibliography	123

Chapter 1

Introduction

Partial differential equations are used to describe a wide variety of physical phenomena such as fluid dynamics, electromagnetism, elasticity, quantum mechanics, or heat conduction. Virtually all problems of practical interest lack analytical solutions, often provably, so numerical solvers are critical to vast fields of engineering and scientific research, including weather and climate prediction. Since the mathematical equations relate continuous functions, the solution space for any particular problem is infinite. However, digital computers can only represent a finite set of states. Hence the necessity to *discretise* partial differential equations: computers may solve the discrete problem which provides an approximate solution to the original continuous equations.

The most common numerical methods to discretise partial differential equations are finite difference methods, finite volume methods, and the finite element method. Finite element methods have solid mathematical underpinnings that make it easy to tune different aspects of the discretisation to the needs of a particular problem. The downside is the complexity of the implementation compared to finite differences, or even finite volume methods. Therefore, simulations for practical purposes often use mathematically inferior methods, while research in advanced finite element techniques is often limited to simple equations to mitigate the high cost of developing a functional implementation.

A key to tackling the complexity of implementation is keeping the software code in close resemblance of the mathematics: a stack of abstractions with relatively simple relations between layers. Although a naïve application of this idea likely leads to poor performance, code generation approaches *can* eliminate the run-time cost of abstraction. The finite element method is particularly suited to code generation since the entire mathematical specification of a finite element problem can be expressed at a high level

of abstraction. The specification of the equations, along with the discrete function space from which each variable is to be drawn, is sufficient to characterise the problem completely.

Code generation based software tools have become an increasingly popular way of implementing complicated numerical solvers using the finite element method. Examples of such software platforms include FreeFem++ [Hecht, 2012], FEniCS [Logg et al., 2012a; Alnæs et al., 2015], and Firedrake [Rathgeber et al., 2016]. These software packages enable high-productivity development of reasonably efficient numerical codes.

The need for high-performance computing contributes another layer of complexity, especially considering modern hardware architectures, which exhibit a growing gap between processor and memory speeds. Spectral methods, for example, are well suited to such computers. High-order finite element methods, such as the spectral element method [Patera, 1984], combine the convergence properties of spectral methods with the geometric flexibility of traditional linear FEM. However, the even greater complexity of these techniques has further limited their uptake in many application domains, especially outside academia [Cantwell et al., 2015].

The key to effective code generation is the application of transformations, especially optimisations, at the right level of abstraction. The same transformation tends to be simpler at higher levels of abstraction, until it becomes inexpressible as the difference between the original and the optimised representations becomes hidden beneath an abstraction. This thesis presents a novel, multi-stage code generation framework for the finite element method. The utility of this approach is evidenced by the capability to generate *sum factorised* code for finite element assembly on quadrilateral and hexahedral elements, thus achieving optimal algorithmic complexity for high-order finite element discretisations.

1.1 Thesis statement

Code generation enables the productive development of efficient implementations of state-of-the-art finite element solvers for problems of practical interest.

1.2 Technical contributions

This work is accompanied by manifold contributions to Firedrake, an automated system for the portable solution of partial differential equations using the finite element method.

Firedrake is a publicly available¹ open source software. Known users include many researchers in numerical mathematics, especially in the domain of geophysical fluid flows. The work described in this thesis is implemented in Firedrake. This includes:

- *Support for unstructured quadrilateral meshes.* This is made composable with the existing extruded mesh functionality, which enables the use of semi-structured hexahedral meshes. Besides software engineering, the main part of this work was a novel algorithm for consistently orienting quadrilateral meshes (chapter 3).
- The *Two-Stage Form Compiler* (TSFC), which has replaced a modified version of the *FEniCS Form Compiler* (FFC) in Firedrake. Chapters 4 and 7 describe TSFC in detail. The new form compiler was one of the key components for supporting non-affine mesh geometry in a mathematically correct manner.
- GEM (section 4.6), the intermediate representation of TSFC, is a *tensor algebra language* and the medium of many important optimisations.
- FInAT, a *smarter library of finite elements* which is able to express the structure inherent to *some* finite elements (chapter 5), for example, tensor product elements. Full utilisation of tensor product elements needs support for quadrilateral and hexahedral cells in the rest of Firedrake (first bullet point).
- *Form compiler algorithms* that can automatically exploit the structure exposed by FInAT to carry out optimisations (chapter 6) such as *sum factorisation* [Orszag, 1980]. These algorithms are implemented as GEM-to-GEM transformers in TSFC.

1.3 Dissemination

The work described in this thesis has been disseminated through journal publications, workshop and conference presentations, and implemented as software packages released under open source licences. Publications in chronological order:

- Homolya and Ham [2016]. *A Parallel Edge Orientation Algorithm for Quadrilateral Meshes*. SIAM Journal on Scientific Computing 38:5 (2016), S48–S61.
This paper presents distributed-memory parallel algorithms to assign consistent edge orientations to quadrilateral meshes (chapter 3).

¹Download at <http://firedrakeproject.org>

- Homolya, Mitchell, Luporini, and Ham [2017b]. *TSFC: a structure-preserving form compiler*. Submitted to SIAM Journal on Scientific Computing (May 2017). This paper introduces TSFC and its main ideas, explains its basic operation, and describes its algorithms (chapters 4 and 7).
- Homolya, Kirby, and Ham [2017a]. *Exposing and exploiting structure: optimal code generation for high-order finite element methods*. Submitted to ACM Transactions on Mathematical Software (November 2017). This paper introduces FInAT, a smarter library of finite elements that can express structure within finite elements (chapter 5), and describes several form compiler algorithms for exploiting the exposed structure (chapter 6).

Talks given at workshops and conferences:

- *Quadrilateral mesh support in Firedrake*. FEniCS Workshop, London, 2015.
- *TSFC: a structure-preserving form compiler*. FEniCS Workshop, Oslo, Norway, 2016.
- *TSFC: a structure-preserving form compiler*. Higher-order DG methods and finite element software for modern architectures, Bath, 2016.
- *TSFC: a structure-preserving form compiler*. PDE Software Frameworks (PDESoft), Coventry, 2016.
- *The Two-Stage Form Compiler, a mechanism for structure-preserving code generation*. SIAM Conference on Computational Science and Engineering (CSE), Atlanta, GA, USA, 2017.
- *Code generation for finite element problems*. Firedrake Workshop, London, 2017.
- *FInAT: automating optimal code generation for high-order finite element methods*. FEniCS Workshop, Luxembourg, 2017.

1.4 Thesis outline

Chapter 2 introduces the finite element method, and presents previous work related to the main contributions of this thesis. It also describes the Firedrake stack and how its components relate to each other. Chapter 3 describes algorithms for the consistent

orientation of quadrilateral meshes. This was key for supporting quadrilateral and hexahedral meshes in Firedrake, which later chapters presuppose. Chapter 4 introduces TSFC and describes its first stage: compilation from UFL to GEM. Chapter 5 introduces FInAT and describes its supported finite elements. Chapter 6 provides algorithms to optimise the intermediate GEM representation, especially to exploit any structure exposed by FInAT. Chapter 7 presents efficient algorithms for code generation from GEM to C. Experimental evaluation is presented in chapter 8. Finally, chapter 9 concludes this thesis and gives some outlook.

Chapter 2

Background

2.1 Preliminaries

As an example for solving partial differential equations using the finite element method, we consider the stationary heat equation

$$-\nabla \cdot (\kappa \nabla u) = f \quad (2.1)$$

on a domain $\Omega \subset \mathbb{R}^d$ with κ thermal conductivity and f heat source, solving for $u : \Omega \rightarrow \mathbb{R}$ temperature field. Suppose the boundary conditions are

$$u = u_0 \quad \text{on } \Gamma_D \subseteq \partial\Omega \text{ and} \quad (2.2)$$

$$\frac{\partial u}{\partial n} = g \quad \text{on } \Gamma_N \subseteq \partial\Omega \quad (2.3)$$

where $\partial u / \partial n$ denotes the derivative in the normal direction. In other words, we have Dirichlet boundary conditions on Γ_D , and Neumann boundary conditions on $\Gamma_N = \partial\Omega \setminus \Gamma_D$.

The solution u of eq. (2.1) shall also satisfy the *variational* or *weak* formulation:

$$\int_{\Omega} -\nabla \cdot (\kappa \nabla u) v \, dx = \int_{\Omega} f v \, dx \quad (2.4)$$

for a suitable test function v . Integrating the left-hand side by parts, applying the divergence theorem, and rearranging terms, eq. (2.4) becomes

$$\int_{\Omega} \kappa \nabla u \cdot \nabla v \, dx = \int_{\Omega} f v \, dx + \int_{\Gamma_N} \kappa g v \, ds. \quad (2.5)$$

For this equation to be well-defined, the *weak derivatives* of u and v must exist on Ω . In other words, they must belong to the Sobolev space $H^1(\Omega)$. Note that the Neumann boundary condition has become part of eq. (2.5); this is called a *natural* boundary condition. The Dirichlet boundary condition, however, has to be imposed on the solution space V and the test function space $\hat{V}$:

$$\hat{V} = \{v \in H^1(\Omega) \mid v = 0 \text{ on } \Gamma_D\}, \quad (2.6)$$

$$V = \{u \in H^1(\Omega) \mid u = u_0 \text{ on } \Gamma_D\}. \quad (2.7)$$

This is called an *essential* boundary condition. We now arrive at the classical variational formulation of eq. (2.1): find $u \in V$ such that

$$a(v, u) = L(v) \quad \forall v \in \hat{V}, \quad (2.8)$$

where a is a bilinear and L is a linear form defined as

$$a(v, u) = \int_{\Omega} \kappa \nabla u \cdot \nabla v \, dx \quad \text{and} \quad (2.9)$$

$$L(v) = \int_{\Omega} f v \, dx + \int_{\Gamma_N} \kappa g v \, ds. \quad (2.10)$$

The finite element method does not discretise the equations, but the solution space. Let $V_h \subset V$ be a finite-dimensional space, and let $\{\psi_i\}_{i=1}^{N_g}$ be its basis. Suppose $u_0 = 0$, so $\hat{V}_h = V_h$. The solution $u \in V_h$ can be written as $u(x) = \sum_{i=1}^{N_g} \mathbf{u}_i \psi_i(x)$ where $\mathbf{u}$ is a vector of the appropriate basis function coefficients. Substituting the test function v with each basis functions reduces eq. (2.8) to a linear system $\mathbf{A} \cdot \mathbf{u} = \mathbf{b}$ with

$$\mathbf{A}_{ij} = a(\psi_i, \psi_j) \quad \text{and} \quad (2.11)$$

$$\mathbf{b}_i = L(\psi_i). \quad (2.12)$$

$\mathbf{A}$ is the bilinear form a assembled as a matrix, and $\mathbf{b}$ is the linear form L assembled as a vector. *Finite element assembly* means this numerical evaluation of matrix $\mathbf{A}$ or vector $\mathbf{b}$ for some given discrete function space V and forms a and L . Then the solution u is obtained by “simply” solving the linear system $\mathbf{A} \cdot \mathbf{u} = \mathbf{b}$. The error of the obtained approximate solution is minimised in the *energy norm*, and orthogonal to the solution space V_h [see Brenner and Scott, 2008, also for further reference].

When using Firedrake, forms are specified in the *Unified Form Language* (UFL) [Alnæs et al., 2014], a domain-specific language embedded in Python. For example:

```

u = TrialFunction(V)
v = TestFunction(V)
a = kappa*dot(grad(u), grad(v))*dx
L = f*v*dx + kappa*g*v*ds

```

In UFL terminology, v and u are called *arguments* of the forms a and L . Forms must always be linear in each of their arguments. For the prescribed spatial functions κ , f , and g , there are several options. For example, they can be specified as analytic expressions: if two homogeneous materials touch at $x_1 = \frac{1}{2}$, then heat conductivity could be

$$\kappa = \begin{cases} 0.75 & \text{if } x_1 \leq \frac{1}{2} \\ 1.21 & \text{if } x_1 > \frac{1}{2} \end{cases}.$$

Let the heat source be $f(x) = 1 - \cos(x_1)$, and the Neumann boundary be isolated $g = 0$. Here x denotes the coordinate vector, thus x_1 is the first coordinate. Note that, while we start indexing from 1, Python starts from 0, so the equivalent UFL would be:

```

x = SpatialCoordinate(domain)
kappa = conditional(x[0] <= 0.5, 0.75, 1.21)
f = 1 - cos(x[0])
g = 0

```

While analytic expressions are useful for simple test cases, in practice, parametrising spatial functions are often represented as functions from finite element function spaces. For example, $f(x) = \sum_{i=1}^{N_f} \mathbf{f}_i \phi_i(x)$ where $\{\phi_i\}_{i=1}^{N_f}$ is the basis of a function space W on Ω , and $\mathbf{f}$ is a vector of basis function coefficients. In UFL terminology, such finite element fields are called *coefficients* of the multilinear form, and forms need not be linear in their coefficients.

Matrix-free methods Large systems of linear equations, such as those resulting from finite element problems, are generally sparse and solved using various *Krylov subspace methods*, which do not strictly require *matrix assembly*, as they only directly use the action of the operator. (Many preconditioners and direct solvers require matrix entries of the assembled operator, however.) Instead of assembling the operator as a sparse matrix, and then applying matrix-vector multiplications, one can also assemble

the operator action on vector $\mathbf{u}$ directly as a parametrised linear form:

$$\begin{aligned} (\mathbf{A} \cdot \mathbf{u})_i &= \sum_{j=1}^{N_g} \mathbf{A}_{ij} \mathbf{u}_j = \sum_{j=1}^{N_g} a(\psi_i, \psi_j) \mathbf{u}_j \\ &= a(\psi_i, \sum_{j=1}^{N_g} \mathbf{u}_j \psi_j) = a(\psi_i, u) \end{aligned} \tag{2.13}$$

where $u(x) = \sum_{j=1}^{N_g} \mathbf{u}_j \psi_j(x)$ is a function isomorphic to the vector $\mathbf{u}$. This approach is commonly known as a *matrix-free method*. High-order discretisations make matrix-free methods very attractive, especially with fast, sum factorised assembly. We cover these topics in more details later in this thesis.

Mixed formulation of partial differential equations

We now consider a mixed formulation of the stationary heat equation, introducing the negative heat flux $\sigma = \kappa \nabla u$ as an auxiliary vector-valued variable, so eq. (2.1) becomes:

$$\sigma - \kappa \nabla u = 0 \tag{2.14}$$

$$\nabla \cdot \sigma = -f \tag{2.15}$$

We search for the solution functions u and σ in suitable function spaces V and Σ respectively. To obtain the weak form, we multiply by arbitrary test functions $\tau \in \Sigma$ and $v \in V$. For simplicity, suppose $u = 0$ on the boundary, and $\kappa \equiv 1$. Then we integrate over Ω , apply integration by parts and the divergence theorem, so the weak form becomes

$$\int_{\Omega} \sigma \cdot \tau + u \nabla \cdot \tau + v \nabla \cdot \sigma \, dx = - \int_{\Omega} f v \, dx. \tag{2.16}$$

This weak formulation needs $V \subset L^2(\Omega)$ and $\Sigma \subset H(\text{div}; \Omega)$, so the *weak divergences* of σ and τ are well defined. $L^2(\Omega)$ denotes the space of square-integrable functions on Ω , and $H(\text{div}; \Omega)$ the space of functions whose weak divergence exists on Ω . Equation (2.16) may be written in UFL as:

```
sigma, u = TrialFunctions(Sigma * V)
tau, v = TestFunctions(Sigma * V)
a = (dot(sigma, tau) + u*div(tau) + div(sigma)*v)*dx
L = -f*v*dx
```

Assembling such mixed forms produces logically block structured matrices, such as

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}^{(1,1)} & \mathbf{A}^{(1,2)} \\ \mathbf{A}^{(2,1)} & \mathbf{0} \end{bmatrix} \quad (2.17)$$

where $\mathbf{A}^{(1,1)}$, $\mathbf{A}^{(1,2)}$, and $\mathbf{A}^{(2,1)}$ are obtained through assembling the split forms

$$a^{(1,1)}(\tau, \sigma) = \int_{\Omega} \sigma \cdot \tau \, dx, \quad (2.18)$$

$$a^{(1,2)}(\tau, u) = \int_{\Omega} u \nabla \cdot \tau \, dx, \quad (2.19)$$

$$a^{(2,1)}(v, \sigma) = \int_{\Omega} v \nabla \cdot \sigma \, dx \quad (2.20)$$

respectively. Note that $\mathbf{A}^{(1,2)}$ and $\mathbf{A}^{(2,1)}$ are generally rectangular, since the trial and test functions of $a^{(1,2)}$ and $a^{(2,1)}$ are from different function spaces.

2.1.1 From global assembly to local assembly

Let $\mathcal{T}$ be a *tesselation* of the domain Ω , so integrals can be evaluated *cellwise* according to $\int_{\Omega} \cdot \, dx = \sum_{K \in \mathcal{T}} \int_K \cdot \, dx$.¹ Typically, each global basis function ψ_i is supported only on a few neighbouring cells, which leaves the $N_g \times N_g$ global matrix $\mathbf{A}$ sparse. Following Kirby and Logg [2006, §2.2], let $\{\psi_i^K\}_{i=1}^{N_l}$ be the restriction to K of the subset of $\{\psi_i\}_{i=1}^{N_g}$ supported on the cell K . Because of local support, $N_l \ll N_g$. We define the $N_l \times N_l$ local matrix $\mathbf{A}^K$ such that

$$\mathbf{A}_{ij}^K = a_K(\psi_i^K, \psi_j^K) \quad (2.21)$$

where $a_K(v, u) = \int_K \kappa \nabla u \cdot \nabla v \, dx$ for the original example. We can assemble $\mathbf{A}$ by assembling $\mathbf{A}^K$ for each K , inserting its entries into the global matrix. This requires a mapping $\iota(\cdot, \cdot)$ from the local numbering of $\{\psi_i^K\}_{i=1}^{N_l}$ to the global numbering of $\{\psi_i\}_{i=1}^{N_g}$, so that ψ_i^K is the restriction to K of $\psi_{\iota(K,i)}$. This process called *global assembly* is described in Algorithm 1.

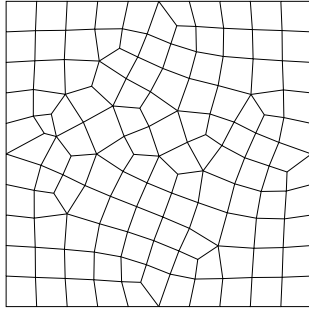
In Firedrake, tessellations of domains, such as Fig. 2.1a, are represented by *mesh* objects. To construct a *function space*, we need a mesh and a *reference element*. Following the notation in Kirby [2014a], the reference element is a finite element on the reference cell $\hat{K}$, and let $\{\Psi_i\}_{i=1}^{N_l}$ denote its basis. The tessellation has to be such that each $K \in \mathcal{T}$ is diffeomorphic to $\hat{K}$ via a mapping $F_K : \hat{K} \rightarrow K$. The global basis is constructed in

¹Boundary integrals such as $\int_{\Gamma_N} \kappa g v \, ds$ in eq. (2.10) can be decomposed analogously.

Algorithm 1 Global assembly [adaptation of Kirby and Logg, 2006, §2.2]

```

1:  $\mathbf{A} \leftarrow \mathbf{0}$ 
2: for all  $K \in \mathcal{T}$  do
3:   Compute  $\mathbf{A}^K$  satisfying eq. (2.21)
4:   for  $i = 1, \dots, N_l$  do
5:     for  $j = 1, \dots, N_l$  do
6:        $\mathbf{A}_{\iota(K,i)\iota(K,j)} \leftarrow \mathbf{A}_{\iota(K,i)\iota(K,j)} + \mathbf{A}_{ij}^K$ 
7:     end for
8:   end for
9: end for
  
```



(a) Quadrilateral mesh on a square domain

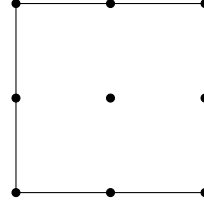
(b) Q_2 finite element

Figure 2.1: Ingredients for constructing a function space

such a manner that its restriction to any cell K is like transferring the reference basis to K , that is

$$\psi_i^K = \Psi_i \circ F_K^{-1}, \quad i = 1, \dots, N_l. \quad (2.22)$$

To ensure continuity between cells in the global basis, each basis function is associated with a topological entity. A basis function associated with a vertex will have support on all cells incident to that vertex, those associated with an edge will have support on all cells incident to that edge, and so on, while basis functions associated with a cell have support on that cell only. Fig. 2.1b shows a Q_2 element on a reference quadrilateral: it has a basis function associated with each topological entity of the reference cell. The construction of the $\iota(\cdot, \cdot)$ local-to-global mapping, such that shared basis functions are treated consistently by all involved cells, is generally nontrivial, especially when some basis functions correspond to tangential or normal evaluations, as is normally the case for $H(\text{div})$ and $H(\text{curl})$ conforming elements. Rognes et al. [2009, §5] solve

this numbering problem for meshes of simplicial cells. Agelek et al. [2017] present solutions for quadrilateral and hexahedral meshes. Homolya and Ham [2016] extend this quadrilateral mesh algorithm for distributed-memory parallelism, which we discuss in detail in chapter 3. Given a local-to-global mapping and the generic global assembly algorithm, we shall now focus on *local assembly*, that is the evaluation of $\mathbf{A}^K$ satisfying eq. (2.21).

2.1.2 Changing to reference space

We continue with a discussion of local assembly, i.e. the evaluation of eq. (2.21). For our original example:

$$\mathbf{A}_{ij}^K = \int_K \kappa \nabla \psi_i^K \cdot \nabla \psi_j^K \, dx. \quad (2.23)$$

It is common practice to change the integral from physical to reference space. This implies changing the integration domain from K to $\hat{K}$ while scaling the integral accordingly by $|\det(J_K)|$ where $J_K = \hat{\nabla} F_K$, and $\hat{\nabla}$ indicates differentiation in reference coordinates $\hat{x}$. Equation (2.22) shows the pullback of basis functions, while the pullback of their gradients is obtained through application of the *chain rule*:

$$\nabla \psi_i^K = J_K^{-T} \hat{\nabla} \Psi_i \circ F_K^{-1}, \quad i = 1, \dots, N_l. \quad (2.24)$$

Similarly, $\hat{\kappa} = \kappa \circ F_K$ is the reference version of κ restricted to K . This results in

$$\mathbf{A}_{ij}^K = \int_{\hat{K}} \hat{\kappa} \left(J_K^{-T} \hat{\nabla} \Psi_i \cdot J_K^{-T} \hat{\nabla} \Psi_j \right) |\det(J_K)| \, d\hat{x}. \quad (2.25)$$

Note that if κ is taken from a finite element function space with reference basis $\{\Phi_i\}_{i=1}^{N_r}$, then its reference value can be directly constructed as

$$\hat{\kappa}(\hat{x}) = \sum_{r=1}^{N_r} \kappa_r^K \Phi_r(\hat{x}) \quad (2.26)$$

where κ^K is a vector of the basis function coefficients of κ local to K . We can now define *local assembly* as the function

$$(F_K, \kappa^K) \mapsto \mathbf{A}^K.$$

Generally speaking, local assembly needs the coordinate mapping F_K , which characterises the local integration domain K , and the local basis function coefficients of all parametrising functions, and it computes the local tensor. Given the weak form as in eq. (2.9), the role of a *form compiler* is to generate low-level code that carries out local assembly.

Mapping $H(\text{div})$ and $H(\text{curl})$ conforming elements

The previous examples have only used the trivial *reference value transformation*:

$$\mathcal{F}(\Phi) = \Phi \circ F_K^{-1} \quad (2.27)$$

which is satisfactory for $H^1(\Omega)$ and $L^2(\Omega)$ Sobolev spaces, but not for $H(\text{div}; \Omega)$ and $H(\text{curl}; \Omega)$ spaces since it does not in general preserve normal or tangential continuity between cells. Instead, one must use the *contravariant Piola mapping*

$$\mathcal{F}^{\text{div}}(\Phi) = \frac{1}{\det(J_K)} J_K \Phi \circ F_K^{-1} \quad (2.28)$$

on $H(\text{div})$ elements to preserve normal continuity, and the *covariant Piola mapping*

$$\mathcal{F}^{\text{curl}}(\Phi) = J_K^{-T} \Phi \circ F_K^{-1} \quad (2.29)$$

on $H(\text{curl})$ elements to preserve tangential continuity. For derivatives, reference transformations can be obtained through the application of the chain rule. With Piola mappings and non-affine cell geometry, this can produce fairly complicated formulas involving second derivatives of F_K . UFL, however, is capable of *symbolic differentiation* to automatically derive them.

Mesh geometry

There are many ways to describe the mesh geometry, represented by $\{F_K\}_{K \in \mathcal{T}}$ in this discussion. The most common approach is to assign coordinates to each vertex of the mesh. In Firedrake, the mesh coordinates are always represented as finite element vector field. The previous example, in fact, is equivalent to a first-order Lagrange space. Such an abstract mesh of triangular cells in a 2-dimensional space is written in UFL as:

```
mesh = Mesh(VectorElement("Lagrange", triangle, 1, dim=2))
```

Setting $\text{dim}=3$ would mean a 2-dimensional surface in a 3-dimensional space, e.g., the surface of a sphere. Setting the degree of the coordinate element higher than one would allow the triangular cells to be curved. Firedrake users, however, typically create a concrete mesh object which extends the UFL class with mesh data that define a tessellation of a concrete physical domain.

2.1.3 Numerical integration and assembly code

The general approach to evaluate integrals, such as over $\hat{K}$ in eq. (2.25), is *numerical quadrature*. Symbolic integration is only feasible in case of a limited set of problems. If the integrand is polynomial, numerical quadrature can be exact up to rounding errors. Let $\{\xi_q\}_{q=1}^{N_q}$ be a set of quadrature points on $\hat{K}$ with corresponding quadrature weights $\{w_q\}_{q=1}^{N_q}$, so – substituting eq. (2.26) – the integral in eq. (2.25) is approximated by

$$\mathbf{A}_{ij}^K = \sum_{q=1}^{N_q} w_q \left(\sum_{r=1}^{N_r} \kappa_r^K \Phi_r(\xi_q) \right) \left(J_K^{-T}(\xi_q) \hat{\nabla} \Psi_i(\xi_q) \cdot J_K^{-T}(\xi_q) \hat{\nabla} \Psi_j(\xi_q) \right) |\det(J_K(\xi_q))|. \quad (2.30)$$

Since we have changed to reference space, basis function evaluations like $\Phi_r(\xi_q)$ and $\hat{\nabla} \Psi_i(\xi_q)$ no longer depend on the cell K , so they can be precomputed. FIAT [Kirby, 2004] implements a wide range of arbitrary-order finite elements, and can produce numerical tensors with the evaluations of basis functions or their derivatives at requested reference points. Specifically, FIAT can provide the numerical tensors Φ and $D\Psi$ such that

$$\Phi_{iq} := \Phi_i(\xi_q) \quad \text{and} \quad (2.31)$$

$$D\Psi_{ik} := \frac{\partial \Psi_i}{\partial \hat{x}_k}(\xi_q). \quad (2.32)$$

Since the coordinate field is represented as an ordinary finite element field, the evaluation of F_K and its Jacobian J_K is analogous to the evaluation of any parametrising function. Let $\{\Theta_i\}_{i=1}^{N_c}$ be the reference basis of the coordinate space and $\mathbf{x}^K$ be the vector of local basis function coefficients, so $J_K(\hat{x}) = \sum_{c=1}^{N_c} \mathbf{x}_c^K \hat{\nabla} \Theta_c(\hat{x})$. Let $D\Theta$ be the tabulation matrix of the coordinate element such that $D\Theta_{iq} := \hat{\nabla} \Theta_i(\xi_q)$. For brevity, we use the following notation for the evaluation of parametrising functions at quadrature points:

$$\bar{\kappa}_q := \sum_{r=1}^{N_r} \kappa_r^K \Phi_{rq} \quad (2.33)$$

$$\bar{J}_q := \sum_{c=1}^{N_c} \mathbf{x}_c^K D\Theta_{cq} \quad (2.34)$$

where $\bar{\kappa}_q \in \mathbb{R}$ and $\bar{J}_q \in \mathbb{R}^{d \times d}$. Equation (2.30) thus becomes

$$\mathbf{A}_{ij}^K = \sum_{q=1}^{N_q} w_q \bar{\kappa}_q \left(\bar{J}_q^{-T} D\Psi_{iq} \cdot \bar{J}_q^{-T} D\Psi_{jq} \right) |\det(\bar{J}_q)|. \quad (2.35)$$

Algorithm 2 presents a crude attempt to turn this equation into a program. This code stores $\bar{\kappa}_q$ and $\bar{J}_q$ in temporary variables, but there is still room for further optimisations. For example, the calculation of $|\det(\bar{J}_q)|$ could be hoisted out of the i, j loops, or the expression in line 9 could be rearranged as

$$\bar{\kappa}_q w_q |\det(\bar{J}_q)| \bar{J}_q^{-1} \bar{J}_q^{-T} D\Psi_{iq} \cdot D\Psi_{jq}$$

and all of $\bar{\kappa}_q w_q |\det(\bar{J}_q)| \bar{J}_q^{-1} \bar{J}_q^{-T} \in \mathbb{R}^{d \times d}$ could be hoisted out of the basis function loops. This pseudo-code also uses a tensor notation: further lowering to scalars is required to generate C code, for example. Chapter 7 gives a detailed discussion on C code generation for tensor algebra expressions such as that defined in eqs. (2.33) to (2.35).

Algorithm 2 Local assembly routine for the bilinear form in eq. (2.9)

```

1: function LOCALASSEMBLY( $\mathbf{x}^K, \boldsymbol{\kappa}^K$ )
2:    $\triangleright$  Precomputed tensors:  $\Phi, D\Theta$ , and  $D\Psi$ 
3:    $\mathbf{A}^K \leftarrow \mathbf{0}$ 
4:   for  $q \leftarrow 1, \dots, N_q$  do
5:      $\bar{\kappa}_q \leftarrow \sum_{r=1}^{N_r} \boldsymbol{\kappa}_r^K \Phi_{rq}$ 
6:      $(\bar{J}_q)_{ab} \leftarrow \sum_{c=1}^{N_c} \mathbf{x}_c^K (D\Theta_{cq})_{ab}$ 
7:     for  $i = 1, \dots, N_l$  do
8:       for  $j = 1, \dots, N_l$  do
9:          $\mathbf{A}_{ij}^K \leftarrow \mathbf{A}_{ij}^K + w_q \bar{\kappa}_q \left( \bar{J}_q^{-T} D\Psi_{iq} \cdot \bar{J}_q^{-T} D\Psi_{jq} \right) |\det(\bar{J}_q)|$ 
10:  return  $\mathbf{A}^K$ 

```

2.1.4 Structure within finite elements

The approach of using FIAT as in eqs. (2.31) and (2.32) enables us to generate code for matrix assembly, however, it also renders certain optimisations infeasible which rely on the structure inherent to some finite elements. This is because FIAT does not, and cannot express such structure, since the tabulations it provides are just numerical arrays, henceforth called *tabulation matrices*.

For example, *sum factorisation* is a well-known technique that drastically improves the assembly performance of high-order discretisations. It relies on being able to write tabulation matrices as a tensor product of smaller matrices. Suppose the reference cell is a square. We take one-dimensional quadrature rules $\{(\xi_{q_1}^{(1)}, w_{q_1}^{(1)})\}_{q_1=1}^{N_{q_1}}$ and

$\{(\xi_{q_2}^{(2)}, w_{q_2}^{(2)})\}_{q_2=1}^{N_{q_2}}$ to define their tensor product with

$$\xi_q = \begin{bmatrix} \xi_{q_1}^{(1)} \\ \xi_{q_2}^{(2)} \end{bmatrix} \quad \text{and} \quad (2.36)$$

$$w_q = w_{q_1}^{(1)} w_{q_2}^{(2)} \quad (2.37)$$

where $q = (q_1, q_2)$. Similarly, we take one-dimensional finite element bases $\{\Phi_{i_1}^{(1)}\}_{i_1=1}^{N_{r_1}}$ and $\{\Phi_{i_2}^{(2)}\}_{i_2=1}^{N_{r_2}}$ to define the tensor product element $\{\Phi_i\}_{i=1}^{N_{r_1}N_{r_2}}$ such that

$$\Phi_i(\hat{x}) = \Phi_{i_1}^{(1)}(\hat{x}_1) \Phi_{i_2}^{(2)}(\hat{x}_2) \quad (2.38)$$

where $i = (i_1, i_2)$, and $\hat{x} = (\hat{x}_1, \hat{x}_2)$. Consequently, $\Phi_i(\xi_q) = \Phi_{i_1}^{(1)}(\xi_{q_1}^{(1)}) \Phi_{i_2}^{(2)}(\xi_{q_2}^{(2)})$, so the relationship between tabulation matrices is

$$\Phi_{iq} = \Phi_{i_1 q_1}^{(1)} \Phi_{i_2 q_2}^{(2)}. \quad (2.39)$$

In other words, the tabulation matrix of the tensor product element is the tensor – or, more specifically, Kronecker – product of the tabulation matrices of the one-dimensional elements. This structure admits more efficient computations of eq. (2.33) since

$$\bar{\kappa}_q = \sum_{r_1=1}^{N_{r_1}} \sum_{r_2=1}^{N_{r_2}} \kappa_{(r_1, r_2)}^K \Phi_{r_1 q_1}^{(1)} \Phi_{r_2 q_2}^{(2)} = \sum_{r_1=1}^{N_{r_1}} \Phi_{r_1 q_1}^{(1)} \sum_{r_2=1}^{N_{r_2}} \kappa_{(r_1, r_2)}^K \Phi_{r_2 q_2}^{(2)} = \sum_{r_2=1}^{N_{r_2}} \Phi_{r_2 q_2}^{(2)} \sum_{r_1=1}^{N_{r_1}} \kappa_{(r_1, r_2)}^K \Phi_{r_1 q_1}^{(1)}. \quad (2.40)$$

This transformation is known as *sum factorisation* [Orszag, 1980], and in chapter 6 we will provide algorithms for its automatic application.

However, it shall be clear that FIAT can only provide Φ_{iq} as a numerical matrix, and cannot express the product structure, rendering sum factorisation infeasible. Therefore, in chapter 5 we will present FInAT as a smarter alternative.

2.1.5 Discontinuous Galerkin methods

Finally, we consider the advection equation

$$\frac{\partial q}{\partial t} + (u \cdot \nabla)q = 0 \quad (2.41)$$

with suitable initial and boundary conditions in a domain $\Omega \subset \mathbb{R}^d$, where $u : \Omega \rightarrow \mathbb{R}^d$ is a prescribed velocity field that carries some substance whose density is denoted by

the scalar field $q(x, t)$. This time we look for a solution q in a space of *discontinuous* functions V . The weak form of the continuous equation in each cell K is

$$\int_K \phi^K \frac{\partial q}{\partial t} dx + \int_K \phi^K (u \cdot \nabla) q dx = 0 \quad (2.42)$$

where $\phi^K \in V_K$ denotes the basis functions on K . Integration by parts and application of the divergence theorem then gives

$$\int_K \phi^K \frac{\partial q}{\partial t} dx = \int_K q \nabla \cdot (\phi^K u) dx - \int_{\partial K} \phi^K q u \cdot n dS \quad (2.43)$$

where n is the outward-pointing unit normal.

In this introduction we will not consider the initial and boundary conditions, nor the choice of a time-stepping scheme to solve this time-dependent partial differentiation equation. Instead, we solely focus on the assembly of the facet² integral term

$$\int_{\partial K} \phi^K q u \cdot n dS. \quad (2.44)$$

Note that q is discontinuous, therefore not defined on the integration domain ∂K . However, approaching ∂K from either inside or outside has a well-defined limit. This generally means that we need data from other cells. Rather than iterating over all cells, and then over all facets of each cell, Ølgaard, Logg, and Wells [2008] have adopted the convention of iterating over all facets, while at the same time considering both cells incident to that facet. This implies that local assembly involves a *macro cell* K consisting of two adjacent cells K^+ and K^- as in Fig. 2.2, while the integration domain is the shared facet Γ_{int} . The two sides of each interior facet are arbitrarily marked with $+$ and $-$. Users have to formulate weak forms without assuming that these signs carry any special meaning.

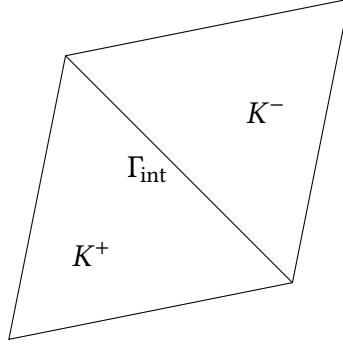
For the simplicity of the upcoming UFL specification, we use the midpoint rule for q along the discontinuity, that is $\tilde{q} = \frac{1}{2}(q^+ + q^-)$. Then eq. (2.44) can be reformulated as

$$\sum_{K \in \mathcal{T}} \int_{\partial K} \phi^K \tilde{q} u \cdot n dS = \sum_{\Gamma_{\text{int}}} \left(\int_{\Gamma_{\text{int}}} \phi^+ \tilde{q} u \cdot n^+ dS + \int_{\Gamma_{\text{int}}} \phi^- \tilde{q} u \cdot n^- dS \right) + \dots \quad (2.45)$$

$$= \sum_{\Gamma_{\text{int}}} \left(\int_{\Gamma_{\text{int}}} \frac{1}{2}(q^+ + q^-) u \cdot (\phi^+ n^+ + \phi^- n^-) dS \right) + \dots \quad (2.46)$$

where the ellipsis corresponds to boundary terms implicitly included in the right-hand side. The interior facet integral is therefore written in UFL as:

²By *facet* we mean a mesh entity of codimension 1. In two dimensions these are the cell edges, and in three dimensions, the cell faces.

Figure 2.2: Two cells K^+ and K^- sharing an interior facet Γ_{int} .

$$\text{avg}(q) * \text{dot}(u, \text{phi}('+') * n('+') + \text{phi}('-') * n('-')) * dS$$

where $n = \text{FacetNormal}(\text{domain})$, q is a discontinuous coefficient, and phi is a discontinuous test function. Utility functions such as

$$\text{jump}(u) = u^+ \cdot n^+ + u^- \cdot n^- \quad \text{and} \quad (2.47)$$

$$\text{avg}(u) = \frac{1}{2} (u^+ + u^-) \quad (2.48)$$

are provided for convenience, however, the only additional UFL primitive is *restriction* which is either $+$ or $-$.

Suppose $\{\phi_i^+\}_{i=1}^{N_l}$ is the local basis on K^+ , and similarly $\{\phi_i^-\}_{i=1}^{N_l}$ is the local basis on K^- . Now let $\{\phi_i\}_{i=1}^{2N_l}$ be a local basis on the macro cell $K = K^+ \cup K^-$ by the following construction:

$$\phi_i(x) = \begin{cases} \phi_i^+(x) & \text{if } 1 \leq i \leq N_l \text{ and } x \in K^+, \\ 0 & \text{if } 1 \leq i \leq N_l \text{ and } x \in K^-, \\ 0 & \text{if } N_l + 1 \leq i \leq 2N_l \text{ and } x \in K^+, \\ \phi_{i-N_l}^-(x) & \text{if } N_l + 1 \leq i \leq 2N_l \text{ and } x \in K^-. \end{cases} \quad (2.49)$$

This construction ensures that any local tensor $\mathbf{A}_{ij}^K = a_K(\phi_i, \phi_j)$ becomes a *block tensor*:

$$\mathbf{A}^K = \begin{bmatrix} \mathbf{A}^{(K^+, K^+)} & \mathbf{A}^{(K^+, K^-)} \\ \mathbf{A}^{(K^-, K^+)} & \mathbf{A}^{(K^-, K^-)} \end{bmatrix} \quad (2.50)$$

This gives an approach for constructing the interior facet tensor $\mathbf{A}^K$ by computing its subtensors. The subtensor $\mathbf{A}^{(K^+, K^+)}$ can be seen as the assembly of a corresponding form $a_K^{(+, +)}$, and the same applies for the other subtensors.

2.2 Previous work on form compilers

A *form compiler* takes a high-level specification of partial differential equations in the weak form, and generates low-level code for local assembly. Automated finite element assembly from a high-level formulation has been employed by various software projects, such as GetDP [Dular et al., 1998], Analysa [Bagheri and Scott, 2003], Sundance [Long et al., 2010], and FreeFem++ [Hecht, 2012]. Kirby and Logg [2006] refer to Analysa as “a precursor to the FEniCS project.” Firedrake [Rathgeber et al., 2016] has adopted the user-level interface and language of FEniCS [Logg et al., 2012a; Alnæs et al., 2015].

The *FEniCS Form Compiler* (FFC) [Logg et al., 2012b] initially generated a *tensor representation* [Kirby and Logg, 2006, 2007] which computes the local tensor as the contraction of a precomputable *reference tensor* with a cell-dependent *geometry tensor*. This is only possible for affine cells where the Jacobian is constant. We demonstrate this on eq. (2.35). Substituting eq. (2.33) and noting that the Jacobian does not depend on the quadrature point, we get

$$\mathbf{A}_{ij}^K = \sum_{q=1}^{N_q} w_q \left(\sum_{r=1}^{N_r} \boldsymbol{\kappa}_r^K \Phi_{rq} \right) \left(\bar{J}^{-T} D\Psi_{iq} \cdot \bar{J}^{-T} D\Psi_{jq} \right) |\det J| \quad (2.51)$$

$$= |\det J| \sum_{r=1}^{N_r} \boldsymbol{\kappa}_r^K \sum_{q=1}^{N_q} w_q \Phi_{rq} \left(\bar{J}^{-1} \bar{J}^{-T} D\Psi_{iq} \cdot D\Psi_{jq} \right) \quad (2.52)$$

$$= |\det J| \sum_{r=1}^{N_r} \boldsymbol{\kappa}_r^K \sum_{q=1}^{N_q} w_q \Phi_{rq} \sum_{k=1}^d \sum_{l=1}^d \left(\bar{J}^{-1} \bar{J}^{-T} \right)_{kl} D\Psi_{iqk} D\Psi_{jqk} \quad (2.53)$$

$$= |\det J| \sum_{r=1}^{N_r} \boldsymbol{\kappa}_r^K \sum_{k=1}^d \sum_{l=1}^d \left(\bar{J}^{-1} \bar{J}^{-T} \right)_{kl} \sum_{q=1}^{N_q} w_q \Phi_{rq} D\Psi_{iqk} D\Psi_{jqk}. \quad (2.54)$$

After this rearrangement, we find that $\sum_q$ acts only on compile-time constants, so we can eliminate the quadrature loop by introducing the *reference tensor* $\mathbf{R} \in \mathbb{R}^{N_l \times N_l \times d \times d \times N_r}$:

$$\mathbf{R}_{ijklr} = \sum_{q=1}^{N_q} w_q \Phi_{rq} D\Psi_{iqk} D\Psi_{jqk}. \quad (2.55)$$

This approach was demonstrated to be more efficient for simple forms than the standard quadrature-based approach, often by a large factor. FFC relies on the *FInite element Automatic Tabular* (FIAT) [Kirby, 2004] to tabulate finite element basis functions at quadrature points. Later, FFC was extended to support *discontinuous Galerkin* methods

[Ølgaard et al., 2008], $H(\text{div})$ and $H(\text{curl})$ conforming elements [Rognes et al., 2009], and immersed manifolds [Rognes et al., 2013]. The increasing capabilities have accrued interest in code generation beyond what the tensor representation is suitable for.

Some complicated non-linear forms simply have no tensor representation: the local tensor cannot be written as the contraction of a reference tensor with a geometry tensor. Furthermore, the size of the reference tensor grows exponentially with the number of parametrising functions in the form, rendering this representation inefficient or even infeasible for a wide variety of problems. To overcome these issues, Ølgaard and Wells [2010] added a *quadrature representation* to FFC, which generates code that performs run-time quadrature, akin to what one would manually write.

Meanwhile, Alnæs and Mardal [2010] developed the *SyFi Form Compiler* (SFC) which was capable of *analytically integrating* suitable forms. Based on the input languages of FFC and SFC, Alnæs et al. [2014] created the *Unified Form Language* (UFL). The FEniCS Project has thus separated the *form language* from the form compilers. Firedrake has also adopted UFL as its form language. As a form compiler, Firedrake initially used a modified version of quadrature representation of FFC.

Luporini et al. [2015, 2017] made significant contributions to the automatic optimisation of finite element assembly kernels, and their work was implemented in COFFEE, which works on a representation that is basically an *abstract syntax tree* (AST) for C code. Accordingly, the modified FFC in Firedrake was made to generate this AST, so that COFFEE can optimise the generated kernels. COFFEE performs both low-level optimisations, such as alignment and padding to facilitate vectorisation by the host C compiler, and high-level optimisations to reduce the operation count. The latter require restructuring the loops and expressions. In essence, COFFEE seeks to determine the optimal trade-off amongst common subexpressions elimination, loop-invariant code motion, and vectorisation.

This modified FFC began to expose its limitations with the introduction of non-affine cells [McRae et al., 2016; Homolya and Ham, 2016]. Initially, non-simplicial cells were supported with an affine approximation: the Jacobian of the coordinate transformation for each cell was approximated by its value at the midpoint. This approximation, however, prevents higher-order convergence on non-affine cells, and similar limitations apply on manifolds with first-order cells [Holst and Stern, 2012]. Furthermore, the quadrature representation suffers from an implementation limitation that expressions are seen as a tree rather than a *directed acyclic graph* (DAG). This can dramatically increase the perceived size of the input expression (see Fig. 2.3), which is detrimental

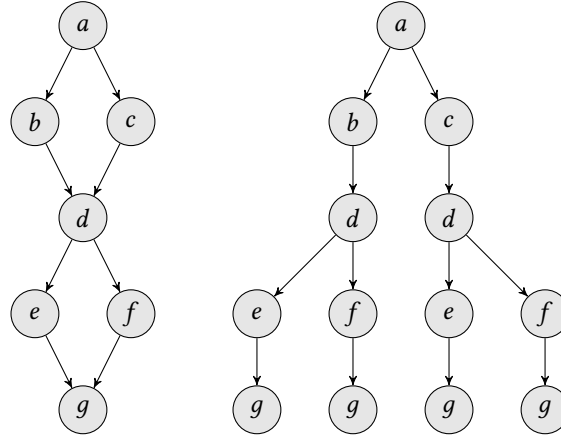


Figure 2.3: Graph representation of an expression with shared subexpressions (left), and a *tree view* of the same expression (right).

to both form compilation speed and generated code size. The effect can be prohibitive for certain forms, especially those derived through *symbolic differentiation* which can produce numerous nested shared subexpressions. This happens to be the case for various formulations of hyperelastic models, as well as when correctly changing to reference space with non-affine geometry.

Martin S. Alnæs and others developed the *UFL Analyser and Compiler System* (UFLACS) [2016] to overcome these limitations. Their work included a new symbolic geometry pipeline in UFL that utilises *automatic differentiation* (AD) to correctly carry out pullbacks even with non-affine geometry. In addition, UFLACS features better expression traversal algorithms that avoid the issue of repeatedly processing shared subexpressions, thus it is suitable to be used with highly complicated forms. Recently, *pre-integration* was added as an optimisation which makes UFLACS competitive in the most common use cases of the tensor representation. UFLACS is integrated as another representation in FFC, and has become its default representation.

2.3 High-order finite elements in numerical software

Code generation based software packages, such as those mentioned on page 2, enable high-productivity development of reasonably efficient numerical codes. They are most commonly used for low-order finite element discretisations of complicated PDEs, while code for the assembly of high-order finite element discretisations has been

predominantly manually written. High-order finite element techniques have been successfully employed in a number of research packages: NEK5000 [Fischer et al., 2008], Nektar++ [Cantwell et al., 2015] and PyFR [Witherden et al., 2014] for computational fluid dynamics, HERMES [Vejchodský et al., 2007] for Maxwell’s equations, EXA-DUNE [Bastian et al., 2014] for porous media flows, pTatin3D [May et al., 2014] for lithospheric dynamics. Hientzsch [2001] implemented an efficient discretisation of Maxwell’s equations using $H(\text{curl})$ conforming spectral elements. However, according to Cantwell et al. [2015], the implementational complexity of these techniques has limited their uptake in many application domains, especially outside academia. Recently, Bastian et al. [2016] report on the utilisation of modern hardware architectures for finite element problems in EXA-DUNE. Kronbichler and Kormann [2017] demonstrate good utilisation of modern CPUs for certain matrix-free discontinuous Galerkin operators. Müthing et al. [2017] present similar achievements on a convection-diffusion-reaction model problem, and anticipate a future paper “on a Python-based code generation tool that can transform a very abstract description of the weak form into highly optimised code.”

2.4 Firedrake overview

Since the contributions of this thesis were implemented software packages that are integrated in Firedrake, we here give an overview of Firedrake’s architecture and its components. One of the motivations for Firedrake is *performance portability* which is reflected in Fig. 2.4, a schematic layout of Firedrake’s components and their relations.

The FEniCS language

FEniCS [Logg et al., 2012a; Alnæs et al., 2015] is a computing platform for the solution of partial differential equations (PDEs) using the finite element method. FEniCS has a high-level Python interface for numerous library features such as reading meshes, defining function spaces and finite element fields, setting up boundary conditions, or invoking solvers. It also includes the Unified Form Language (UFL) [Alnæs et al., 2014] for defining the weak forms of PDEs. This interface and UFL together form what we call *the FEniCS language*: the language which users of FEniCS have to use to develop numerical solvers for some particular PDEs.

Firedrake is a different implementation of the FEniCS language. It includes the same UFL component as FEniCS, and implements more or less the same Python interface. To

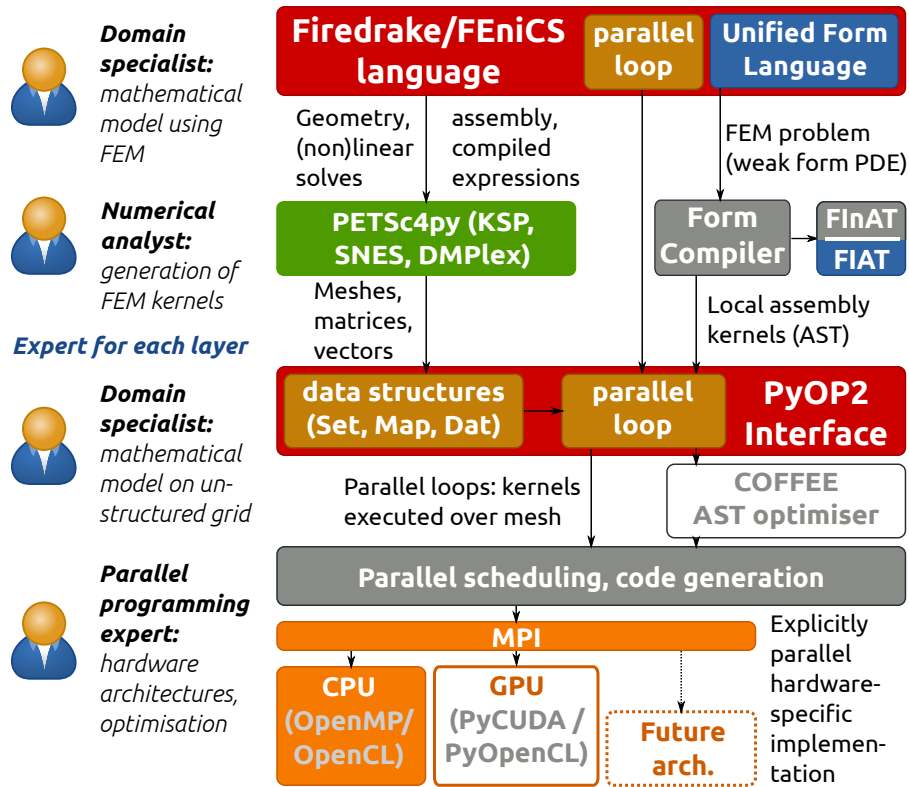


Figure 2.4: The architecture diagram of Firedrake, its abstraction layers and main components. Components shared with FEniCS are in blue, while the tools adopted from PETSc are in green. Interface layers are represented in red, and our code generation components are in grey. Adapted from Rathgeber et al. [2016, Fig. 1].

specify weak forms, UFL only defines abstract mesh objects, while FEniCS and Firedrake provide concrete mesh implementations. For mesh representation, Firedrake relies on DMPLex [Knepley and Karpeev, 2009; Lange et al., 2016], a PETSc module.

Libraries of finite elements: FIAT and FInAT

Firedrake offers a wide range of finite elements implemented in the *FInite element Automatic Tabulator* (FIAT). Kirby [2004] implemented many arbitrary-order simplicial finite elements, including continuous, discontinuous, $H(\text{div})$ and $H(\text{curl})$ conforming elements. McRae et al. [2016] extended FIAT with tensor product elements, and with element algebra enabling construction of $H(\text{div})$ and $H(\text{curl})$ conforming elements on tensor product cells.

In this thesis we will show that FIAT, however, provides an interface that renders certain optimisations infeasible that are critical especially for high-order discretisations. To resolve this issue, Homolya et al. [2017a] introduce FInAT, a smarter library of finite elements, which we describe in chapter 5. FInAT creates a layer on top of FIAT: it presents a more abstract interface, nevertheless it does delegate the implementation of *some* elements to FIAT.

Form compiler

Firedrake originally incorporated a modified version of the *FEniCS Form Compiler* (FFC): the *quadrature representation* of FFC was ported for use in Firedrake. FFC makes use of FIAT to tabulate finite elements.

Because of the limitations of FFC, as discussed in section 2.2, Homolya et al. [2017b] have developed the *Two-Stage Form Compiler* (TSFC) to replace FFC in the Firedrake toolchain. Initially, TSFC used FIAT to tabulate finite elements, but later switched to FInAT to exploit the structure inherent to *some* finite elements in an automatic code generation setting [Homolya et al., 2017a]. The ideas, techniques, and algorithms implemented in TSFC and FInAT constitute the main body of this thesis.

COFFEE remained crucial for optimisations even after switching from FFC to TSFC, however, it is currently being phased out: Tianjiao Sun has ported a simplified version of Luporini et al. [2017] to operate on the internal representation of TSFC instead.

Iteration over the mesh graph: PyOP2

PyOP2 [Markall et al., 2013; Rathgeber et al., 2012] instruments the execution of *local kernels* over an unstructured mesh. In case of finite element problems, these local kernels are most typically responsible for *local assembly*. PyOP2 generates wrapper code for *global assembly*, takes care of necessary halo exchanges when distributed-memory parallelism is employed, and also manages dependencies between *parallel loops*. Its concepts are shared with OP2 [Giles et al., 2012], however, PyOP2 is a Python implementation that features run-time code generation and just-in-time compilation.

Linear and non-linear solvers: PETSc

Firedrake uses the linear and non-linear solvers of PETSc [Balay et al., 2017b,a, 1997], and of other solver packages that PETSc has bindings for. Firedrake users have full access to configure the composable solver infrastructure of PETSc. Building upon

this infrastructure, Kirby and Mitchell [2017] provide preconditioners and matrix-free operators that make use of the available symbolic formulation, rather than treating the assembled operator simply as a black box.

Chapter 3

Consistent edge orientations for quadrilateral meshes

Later chapters of this thesis presuppose the ability to solve partial differential equations on quadrilateral and hexahedral meshes. Here we discuss the orientation problem that had to be solved to support unstructured quadrilateral meshes in Firedrake. This support also extends to semi-structured hexahedral meshes in composition with the existing extruded mesh functionality. The content of this chapter is derived from [Homolya and Ham \[2016\]](#).

A characteristic of the finite element method for the solution of partial differential equations is that the representation of functions over the domain can be chosen from a wide range of function spaces. This choice is achieved by selecting a particular reference finite element to be used for each cell in the meshed domain. Information is communicated between adjacent cells either by shared degrees of freedom on the cell boundaries or by flux integrals over the facets between cells. For every case except the lowest degree continuous finite elements, these coupling terms require the orientations of the two cells adjacent to each facet to be reconciled with the orientation of that facet. Failure to do so will result in false identification of degrees of freedom falling on the facets, or incorrect accounting for the flux direction through facets.

Consequently, every finite element implementation which supports facet integrals, elements of polynomial degree greater than one, or $H(\text{div})$ or $H(\text{curl})$ conforming elements must somehow ensure that adjacent cells agree on the orientation of the intervening facet. This can either be achieved by explicitly recording orientation information and employing this in the local and/or global assembly operations, or it

can be achieved by ensuring, in a sense which we will later make formal, that the local orientation of facets relative to each cell in the mesh is consistent with the global orientation of that facet. libMesh [Kirk et al., 2006] uses global vertex numbers to determine the direction of edges for $H(\text{curl})$ conforming elements; however, neither $H(\text{div})$ conforming nor higher-order elements are supported. FreeFem++ [Hecht, 2012] does not support quadrilaterals. Nektar++ [Cantwell et al., 2015] stores and uses explicit per-cell edge orientations, and also supports mixed cell meshes (e.g., mixed triangles and quadrilaterals). These orientations are employed to match degrees of freedom on facets for higher-order elements; however, no sign flipping is necessary since $H(\text{div})$ and $H(\text{curl})$ conforming elements are not supported. Raviart–Thomas [1977] elements in DUNE [Bastian et al., 2006] store explicit signs (-1 or $+1$) for each facet, which are referenced from manually written formulæ.

The consistent orientation approach has the advantage that the integral assembly code is simpler than that required by the explicit orientation approach, at the cost of somehow establishing the consistent orientation, and with the limitation that a consistent orientation may not exist for all meshes. Rognes et al. [2009] give a detailed discussion of the benefits of consistent orientations for the efficient assembly of finite elements. They also provide an algorithm for finding consistent facet orientations for simplicial meshes, which is used in FEniCS and in Firedrake, but this algorithm does not extend to quadrilaterals. However, deal.II [Bangerth et al., 2007] contains an algorithm for constructing consistent orientations for any quadrilateral mesh of an orientable manifold, originally¹ described in the source code documentation [Bangerth et al., 2003].

The algorithm employed in deal.II as a post-processing step after reading the mesh is serial. This is less than ideal on modern supercomputers, for which it would be both more convenient and more efficient to construct the orientation using a distributed parallel algorithm. The core contribution of this chapter is the extension of this algorithm for distributed memory parallel computers, which facilitates its seamless application as part of a parallel simulation system.

The formal definition of the orientation problem is given in section 3.1 and the proof of the existence of a solution in section 3.2. Section 3.3 restates the deal.II serial algorithm in terms of this proof. The algorithm employed in Firedrake is presented in section 3.4, with experimental evaluation in section 3.5. In section 3.6, our analysis establishes a link between the well-known *Union-Find algorithm* and the construction of

¹This has been eventually published in Agelek et al. [2017], which also addresses the problem of consistent orientations for hexahedral meshes.

a consistent orientation of a quadrilateral mesh. Consequently, we explore the existing work on the parallelisation of the Union-Find algorithm since it can be easily adapted to construct further parallel algorithms for mesh orientations.

3.1 Problem statement

We here formally define the problem that we later analyse and provide algorithmic solutions to. We say that a mesh has *consistent facet orientations* if

- (i) facet orientation is *intrinsic* to the facet, i.e., it does not depend on the cell from which the facet is accessed;
- (ii) there is a fixed reference cell such that for each cell in the mesh there exists a mapping to the reference cell under which the orientation of each facet incident to that cell matches the orientation of the corresponding facet of the reference cell.

The orientation of facets on the reference cell can be arbitrary for the purpose of the above definition, although this choice can limit the scope of meshes for which a consistent facet orientation exists. Henceforth we tackle the problem of finding *consistent edge orientations* for quadrilateral meshes on 2-dimensional orientable manifolds.

3.2 Proof of the existence of a solution

We prove that *consistent edge orientations* exist for any quadrilateral mesh on an *orientable* 2-dimensional manifold. Later we build on the analysis below to construct algorithms that find consistent orientations.

As mentioned earlier, the choice of edge orientations on the reference quadrilateral can affect the set of meshes for which consistent orientations exist. Up to symmetry, there are four different edge orientations of a quadrilateral. These are shown in Fig. 3.1. The symmetries arise since we can arbitrarily choose the first vertex and the direction of traversal when defining a mapping to the reference cell. For each case, one can find a mesh on a 2-dimensional manifold which does not have consistent edge orientations. A counterexample for (a) can be found in [Bangerth et al. \[2003\]](#). A structured grid with an odd number of cells in both direction, folded into a torus, is a counterexample for (b) and (d). A structured grid bent into a Möbius strip is a counterexample for (c).

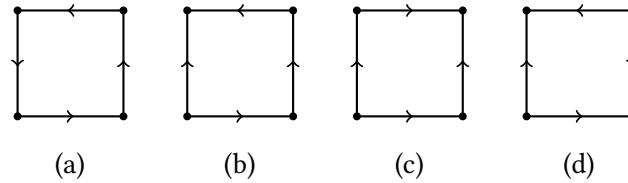


Figure 3.1: All distinct quadrilateral edge orientations, after considering symmetries.

However, when we restrict our attention to meshes on *orientable* manifolds (that is, the vast majority of domains actually employed for finite element problems), then only (c) remains without a counterexample. We now prove existence for this case.

Theorem 3.1. *Consistent edge orientations exist for any quadrilateral mesh on an orientable 2-dimensional manifold, given the edge orientations on the reference quadrilateral as in Fig. 3.1c.*

Proof. Notice that on the reference quadrilateral, opposite edges always have the same orientation. When constructing a global orientation of the cells in a mesh, this implies that fixing the orientation of one edge *determines* the orientation of the opposite edge. Conversely, the orientation of an edge imposes no restriction on the orientation of the two adjacent edges. Since each interior edge of the mesh is adjacent to two cells, fixing the orientation of one edge actually fixes the orientation of every edge reachable from the first edge by an unbroken sequence of opposite cell edges. This defines a relation “the orientation of edge a determines the orientation of edge b .” We will refer to this as the *orientation determination relation*. It is easy to see that this relation is reflexive, symmetric, and transitive, and is therefore an equivalence relation which defines equivalence classes on the set of edges. The form of each equivalence class is a path or ribbon (see Fig. 3.2) through the mesh connecting successive opposite faces. Within each class, the orientation of any edge determines the orientation of all other edges. Conversely, since the orientation of an edge only determines the orientation of the opposite edge and not that of the adjacent edges, separate classes can be oriented independently.

Note that if there exists a consistent edge orientation for a quadrilateral mesh, one can invert the orientation of every edge in any equivalence class and still have a consistent edge orientation. Thus when assigning orientations to a set of dependent edges, we can arbitrarily choose the orientation of one edge, and that orientation propagates to all dependent edges. Either both initial choices are fine or a consistent orientation does not exist.

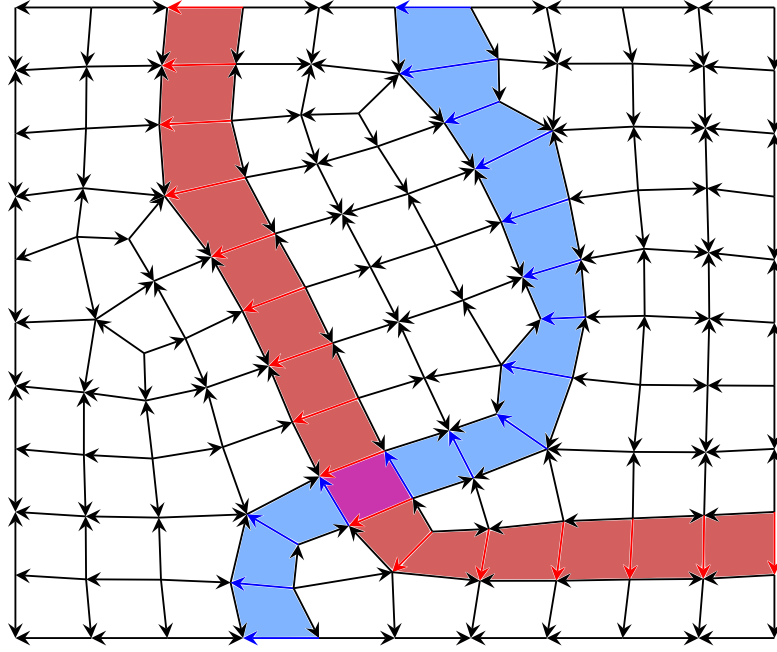


Figure 3.2: Unstructured quadrilateral mesh with consistent edge orientations. Two “ribbons” (sets of edges where the orientation of any edge determines the orientation of all edges) are highlighted in dark red and light blue.

This implies that a consistent orientation will always exist for a mesh, unless there is a set of dependent edges for which the orientations necessarily conflict. That is, there is an edge u , whose orientation forces a different orientation on an edge v through path p_1 than through path p_2 . From a slightly different point of view, there is a cycle

$$p = u \xrightarrow{p_1} v \xrightarrow{p'_2} u$$

around u , where p'_2 is the reverse of p_2 , such that the orientation of u forces a conflicting orientation on itself through path p . Without loss of generality we can assume that p is a simple cycle, i.e., p contains each of its edges only once (except for u , which appears as starting and ending edge in p). In this case the quadrilateral cells in p form a Möbius strip (see Fig. 3.3), which implies that the mesh is not on an orientable manifold.

We have shown that if the mesh does not admit consistent edge orientations, then its manifold is non-orientable. It follows immediately that consistent orientations exist for any quadrilateral mesh on an orientable manifold. $\square$

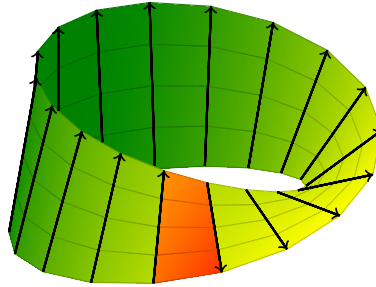


Figure 3.3: Edge orientations propagate in a conflicting manner on a Möbius strip, and conversely, if a cycle propagates orientations in a conflicting way, then connecting the cells will form a Möbius strip.

3.3 Serial algorithm

Based on the above discussion, Algorithm 3 describes the steps of consistently orienting a quadrilateral mesh. We assume that initially $\text{visited}[e] = \text{False}$ and $\text{orientations}[e]$ is undefined for all $e \in E$, where E denotes the set of edges in the mesh. For generality, lines 13 and 14 lack the details of mesh representation and orientation representation.

Lines 10 to 16 run only $|E|$ times, since each time they flip a False to True in visited. Since the quadrilateral mesh is on a 2-dimensional manifold, only 1 or 2 cells can be incident to an edge (line 12); therefore lines 10 to 16 take $O(|E|)$ total runtime, not considering the time spent in recursive ORIENT calls outside these lines. It also follows that ORIENT is called $O(|E|)$ times: $|E|$ times in line 2, and between $|E|$ and $2|E|$ times in line 15. Hence the serial algorithm takes $O(|E|)$ time.

3.4 Parallel algorithm

This section describes the *parallel* extension of the above algorithm for *distributed memory* systems, as it is implemented in Firedrake.

The basic idea of Firedrake’s parallel implementation is to first solve edge orientation locally, then flip some local ribbon segments until all local domains agree on the orientation of all shared edges (see Fig. 3.4). To speed up flipping local ribbon segments, each parallel process records connections between its shared edges. Interior edges are

Algorithm 3 Serial algorithm

```

1: for all  $e \in E$  do
2:   ORIENT( $e, \uparrow$ )
3: end for

4: procedure ORIENT( $e, o$ )
5:   if visited[ $e$ ] then
6:     if orientations[ $e$ ]  $\neq o$  then                                 $\triangleright$  Möbius strip found
7:       abort
8:     end if
9:   else
10:    visited[ $e$ ]  $\leftarrow$  True
11:    orientations[ $e$ ]  $\leftarrow o$ 
12:    for all  $c \in$  cells incident to  $e$  do
13:       $e' \leftarrow$  edge opposite to  $e$  in  $c$ 
14:       $o' \leftarrow$  orientation for  $e'$  consistent with  $e$  having orientation  $o$ 
15:      ORIENT( $e', o'$ )
16:    end for
17:  end if
18: end procedure

```

realigned only when agreement on the orientations of all shared edges is reached.

Algorithm 4 describes how parallel processes negotiate the orientation of shared edges. Before the negotiation starts, each process aligns its edges locally, while populating the `affects_edge` and `affects_orient` arrays. If u and v are edges shared with other processes, and connected by a local ribbon segment, then `affects_edge` is populated with entries `affects_edge[u] = v` and `affects_edge[v] = u`, and `affects_orient` is populated with entries `affects_orient[u] = affects_orient[v] =  $\uparrow$` when u and v have the same orientation, and with entries `affects_orient[u] = affects_orient[v] =  $\downarrow$` otherwise. If u is a shared edge, but it does not connect to any other shared edge, these arrays are not updated. If we encounter a closed loop while locally orienting edges, there is no need to negotiate the orientation its edges with other parallel processes.

Another prerequisite is to initialise the `our_weight` and `our_orient` arrays with the weights and proposed orientations of the shared edges of a parallel process. The

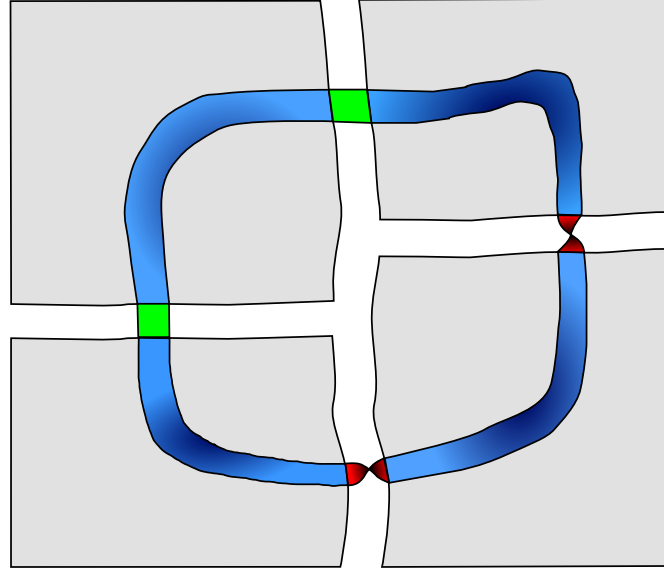


Figure 3.4: A “ribbon” in the global mesh, and its segments in the local domains. It is assumed that the orientation of edges is consistent inside these segments. However, the local domains *may* (green, straight connection) or *may not* (red, twisted connection) agree on the orientation of shared edges.

o	o'	$o \text{ XOR } o'$
↑	↑	↑
↑	↓	↓
↓	↑	↓
↓	↓	↑

Table 3.1: The XOR operator

weights shall be globally unique for all local ribbon segments. We use $\text{our_weight}[e] = \text{size} \cdot l(e) + \text{rank}$, where $l(e)$ is the local index of the ribbon segment which e belongs to, size is the number of parallel processes, and rank is the index of the current process. We need to ensure that locally connected edges have the same weight and consistent orientation initially. This invariant is maintained by Algorithm 4.

Algorithm 4 consists of a main loop which terminates when there are no more conflicts in the proposed orientations for shared edges. In lines 3 and 4 each parallel process exchanges its proposed orientations and weights for its shared edges with its neighbours. their_weight and their_orient have type and size identical to our_weight

Algorithm 4 Parallel algorithm

```

1: conflict  $\leftarrow$  True
2: while conflict do
3:   EXCHANGEEDGEData(our_orient, their_orient)
4:   EXCHANGEEDGEData(our_weight, their_weight)
5:   conflict  $\leftarrow$  False
6:   for all  $e \in$  shared edges do
7:     if our_weight[ $e$ ] = their_weight[ $e$ ] then
8:       if our_orient[ $e$ ]  $\neq$  their_orient[ $e$ ] then ▷ Möbius strip
9:         abort
10:      end if
11:    else if our_weight[ $e$ ] < their_weight[ $e$ ] then
12:      our_weight[ $e$ ] = their_weight[ $e$ ]
13:      our_orient[ $e$ ] = their_orient[ $e$ ]
14:      if  $e \in$  affects_edge then
15:         $e' \leftarrow$  affects_edge[ $e$ ]
16:         $o' \leftarrow$  affects_orient[ $e$ ] XOR our_orient[ $e$ ]
17:        if our_orient[ $e'$ ]  $\neq$   $o'$  then
18:          conflict  $\leftarrow$  True
19:        end if
20:        our_weight[ $e'$ ] = our_weight[ $e$ ]
21:        our_orient[ $e'$ ] =  $o'$ 
22:      end if
23:    end if
24:  end for
25:  conflict  $\leftarrow$  ALLREDUCE(conflict, OR)
26: end while

```

and `our_orient` respectively, but they contain the values proposed by the neighbouring processes. The general rule is to adopt the orientation with the larger weight. Lines 11 to 23 handle the adoption of the remote weight and orientation. Line 14 checks if such an update needs propagation via the local ribbon segment. Lines 15 to 21 propagate the new information to the other connected shared edge, and check if it causes that edge to flip. (The XOR operator used in line 16 is defined by Table 3.1.) If yes, then another round of exchange is necessary, since the orientation for e' that this process forwarded in line 3, has now become outdated. When the local weight is larger than the remote weight, there is nothing to do: the other parallel process will adopt our orientation. Since all weights are initially unique, and then they are copied between neighbours, the local weight being equal to the remote weight means that both parallel processes have their local ribbon segment aligned to the same “source.” Therefore a mismatch in the orientation is possible only if the global mesh contains a Möbius strip, which is checked in lines 8 to 10.

This algorithm terminates in at most k communication rounds, where k is the maximum number of local segments in any ribbon of the global mesh. Since each ribbon is “untwisted” independently, further discussion focuses on one arbitrary ribbon in the global mesh. Any time there is a conflict between the orientation of two local segments, the one with the higher weight “wins”; thus we can conclude that the orientation of a ribbon in the global mesh is ultimately determined by the local segment which has the highest weight. Initially only one segment has this highest weight. In each communication round, one or two additional segments adopt this weight and align their orientations, until the information has propagated to all segments. At that point, there are either no conflicts along the ribbon, or the algorithm will abort in the next communication round due to finding a Möbius strip. Therefore the parallel algorithm terminates in at most k communication rounds.

Theoretically, k can be large with respect to other parameters of the mesh. For example, one can mesh an annulus domain with a very long, spiraling ribbon. However, a far more typical scenario is for each ribbon to either enter one side of the domain and exit the other, or to form a loop. In either of these cases, assuming the domain has a bounded aspect ratio and that the parallel decomposition approximately minimises the surface area of partitions, k will be $O(\sqrt{P})$, where P is the number of parallel processes.

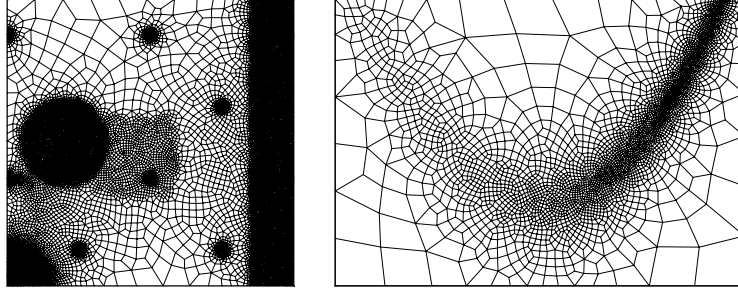


Figure 3.5: Low-resolution version of mesh samples t10 (left) and t11 (right).

3.5 Experiments

We have run several scaling experiments on the ARCHER UK National Supercomputing Service. We used 6 different meshes to evaluate the performance of the parallel algorithm described in section 3.4:

s_square: structured grid on a square domain

s_sphere: cubed sphere mesh (a quadrilateral mesh of the surface of a sphere formed by deforming a cube to sphere shape and refining)

u_square: unstructured mesh on a square domain

u_sphere: unstructured mesh on the surface of a sphere

t10, t11: unstructured meshes on a rectangular domain with nonuniform resolution (see Fig. 3.5)

The first two were generated using Firedrake utility functions, and the other four were generated with Gmsh [Geuzaine and Remacle, 2009], version 2.8.3. t10 and t11 are examples from the Gmsh tutorial. To give an impression of the level of irregularity of these meshes, low-resolution versions are presented in Fig. 3.5. We used meshes with cell count in the order of millions; unfortunately mesh generation becomes very difficult for meshes bigger than that.

We used Firedrake to construct consistent edge orientations for each of these meshes on up to 24576 cores (1024 computing nodes, each with 24 CPU cores). We measured the number of required communication rounds (Fig. 3.6) and execution time (Fig. 3.7). Each experiment was repeated 4 times, except for those involving the largest number of cores; for cost efficiency reasons, they were carried out only once. The number of

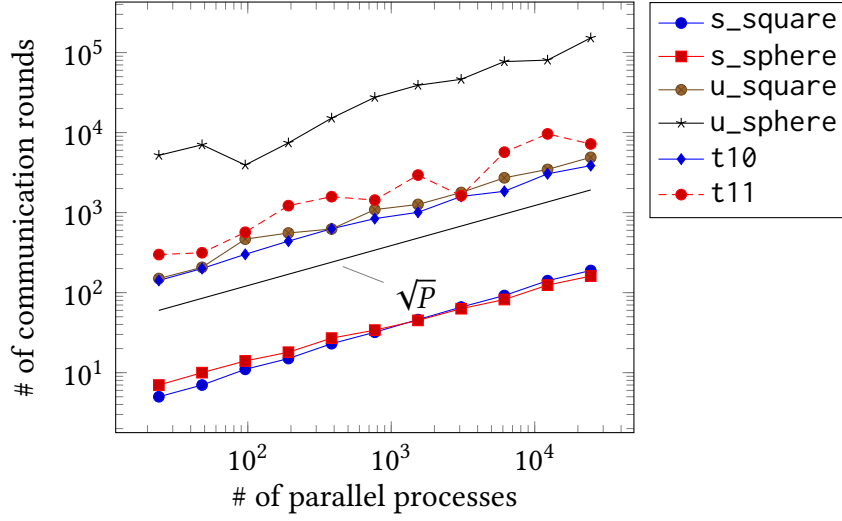


Figure 3.6: Number of required communication rounds for each mesh sample.

communication rounds were the same during each instantiation of the same experiment; however, the execution times varied greatly, especially when they were under a second. This behaviour can have a number of causes on a supercomputing system, such as the allocation of compute nodes, which users generally cannot control. Despite the variation in individual measurements, the data show a consistent trend as the number of parallel processes increases. Fig. 3.7 shows the average of 4 measurements for each data point.

Unsurprisingly, the construction edge orientations for structured meshes takes relatively few communication rounds. Fig. 3.6 also shows that for a rectangular domain, the number of required communication rounds stays the same order of magnitude in case of nonuniform meshing as with uniform resolution. However, the unstructured mesh on a spherical domain requires at least one order of magnitude more communication rounds. This is probably caused by the fact that a sphere does not have a boundary, and thus all ribbons must form a closed loop, so traversing a ribbon must continue until the starting edge is reached again. Regardless, our measurements confirm our predicted $O(\sqrt{P})$ communication rounds for strong scaling. Table 3.2 collects the slopes of lines fitted to the log-log data shown in Fig. 3.6, all values being close to the expected 0.5.

In most of the cases shown in Fig. 3.7, the execution time initially decreases as the number of parallel processes increases. It is likely that this is due to the parallelisation of the local work each process has to do. Eventually the increasing number of communication rounds becomes dominant, and the execution times grow with the number

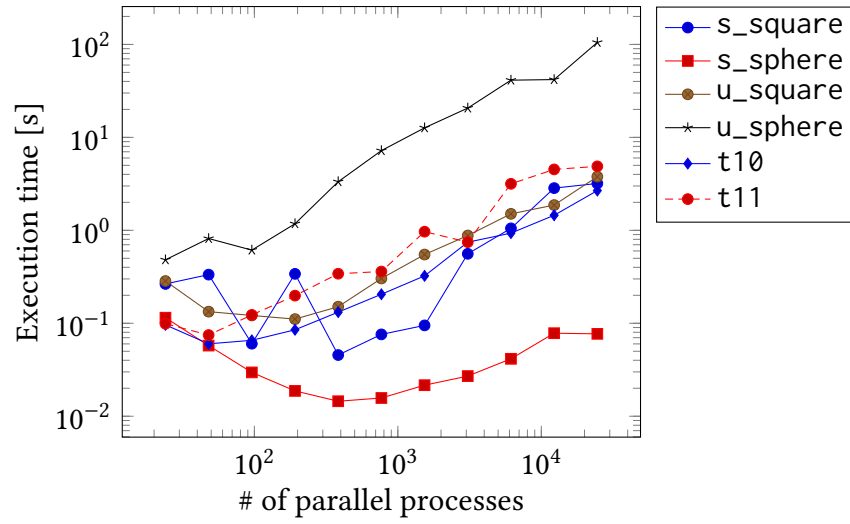


Figure 3.7: Execution time to construct consistent edge orientations in parallel for each mesh sample.

Test case	Slope
u_square	0.49
s_square	0.53
t10	0.47
t11	0.49
s_sphere	0.45
u_sphere	0.53

Table 3.2: Slopes of lines fitted to the log-log data shown in Fig. 3.6.

of parallel processes. However, even the unstructured sphere mesh with almost one million cells, running on about a quarter of ARCHER, takes only 105 seconds. For users running supercomputer-scale simulations, the few minutes spent on edge orientations are not likely to become a bottleneck.

3.6 Alternative approaches to parallelism

The edge orientation problem can be reduced to the well-known *Union-Find algorithm*, also known as the *disjoint set* data structure [Cormen et al., 2001]. We established in section 3.2 that the orientation determination relation is an equivalence relation which defines equivalence classes (the “ribbons”) on the set of edges. To discover the ribbons, we can immediately use the Union-Find algorithm. However, with an extension to it, we can also retrieve the correct edge orientations.

First we briefly introduce the Union-Find algorithm. Let $\Sigma = \{e_1, e_2, \dots, e_n\}$ denote the set of all abstract elements, and let $S_1, S_2, \dots, S_k$ be disjoint nonempty sets over these elements, with $k \leq n$. The sets are initially singletons: $S_i = \{e_i\}, i = 1, \dots, n$. The UNION operation destructively joins two sets. The FIND operation can be used to determine whether two elements, e_i and e_j , are in the same set. Since the sets are disjoint, each element is in exactly one set. Each set is identified by its *representative element*, which can be any of its elements. The FIND operation returns, for any element e_i , the representative element of the set to which e_i belongs. Thus $\text{FIND}(e_i) = \text{FIND}(e_j)$ if e_i and e_j are in the same set.

To solve the edge orientation problem, the edges of the mesh become the abstract elements for the Union-Find algorithm, starting with a singleton set for each edge. Then, for each cell, we join both pairs of opposite edges—more precisely we join the sets to which those opposite edges belong. When done, each set corresponds to a “ribbon,” a set of edges which determine each other’s orientation. However, we still need to traverse the ribbon to determine the permitted relative orientations between edges. We will avoid this traversal by extending the Union-Find algorithm, but let us first briefly present the usual representation of sets and the algorithms for FIND and UNION.

Each element u links to its *parent element* $p(u)$. For representative elements, $p(u) = u$. For all other elements, $p(u)$ is another element in the same set, and following the parent links must eventually lead to the representative element of the set. Algorithms 5 and 6 show the implementation of the FIND and UNION operations, respectively.

FIND simply traverses the parent links until it reaches the representative element.

Algorithm 5 FIND operation for disjoint sets

```

1: function FIND( $u \in \Sigma$ )
2:   if  $p(u) = u$  then
3:     return  $u$ 
4:   else
5:     return FIND( $p(u)$ )
6:   end if
7: end function

```

Algorithm 6 UNION operation for disjoint sets

```

1: procedure UNION( $u \in \Sigma, v \in \Sigma$ )
2:    $r_u \leftarrow$  FIND( $u$ )
3:    $r_v \leftarrow$  FIND( $v$ )
4:   if  $r_u \neq r_v$  then       $\triangleright r_v$  becomes the representative element of the united set
5:      $p(r_u) \leftarrow r_v$ 
6:   end if
7: end procedure

```

A variation of FIND also reassigns the parent links of each element on this path directly to the representative element; this technique is called *path compression*. UNION first calls FIND to look up the representative elements of both sets. If the two sets are indeed different, then it assigns the parent link of one of the representative elements to point to the other representative element, thus effectively merging the sets. Depending on the exact variant of the Union-Find algorithm [Patwary et al., 2010], there is generally a rule to determine which one of r_u and r_v remains a representative element, but we ignore that detail for this discussion. The textbook version of the Union-Find algorithm [Tarjan, 1975] uses *rank-based* merging in UNION and path compression in FIND, and it is proven that m FIND and n UNION operations execute in $O((m + n)\alpha(m, n))$ time, where $\alpha(m, n)$, the functional inverse of Ackermann's function, is so slow-growing that for all practical purposes it can be considered constant.

To address the edge orientation problem, we attach a binary value $\omega(u)$ to each parent link, which describes the permitted relative orientation between the edge u and its parent edge $p(u)$. Algorithms 7 and 8 describe FIND and UNION with orientations, respectively. The changes include the maintenance of orientation, and Möbius strip detection in UNION when one tries to connect a ribbon to itself with the inconsistent

orientation.

Algorithm 7 FIND operation for disjoint sets *with orientation*

```

1: function FIND( $u \in \Sigma$ )
2:   if  $p(u) = u$  then
3:     return  $u, \uparrow$ 
4:   else
5:      $r_u, o_p \leftarrow \text{FIND}(p(u))$ 
6:     return  $r_u, \omega(u) \text{ XOR } o_p$ 
7:   end if
8: end function

```

Algorithm 8 UNION operation for disjoint sets *with orientation*

```

1: procedure UNION( $u \in \Sigma, v \in \Sigma, o \in \{\uparrow, \downarrow\}$ )
2:    $r_u, o_u \leftarrow \text{FIND}(u)$ 
3:    $r_v, o_v \leftarrow \text{FIND}(v)$ 
4:   if  $r_u = r_v$  then
5:     if  $o_u \text{ XOR } o \text{ XOR } o_v = \downarrow$  then ▷ Möbius strip found
6:       abort
7:     end if
8:   else ▷  $r_v$  becomes the representative element of the united set
9:      $p(r_u) \leftarrow (r_v, o_u \text{ XOR } o \text{ XOR } o_v)$ 
10:  end if
11: end procedure

```

This extension can be easily applied to most of the relevant variants of the Union-Find algorithm, including parallel ones. Algorithm 9 describes the algorithm for solving the edge orientation problem using the Union-Find algorithm extended with orientations. While Algorithm 9 itself does not feature any parallel primitives, it will solve the edge orientation problem in parallel if a parallel variant of the Union-Find algorithm is employed. In that case each parallel process only iterates over its *local* cells and edges.

The advantage of this approach is that we can easily reuse existing work on parallelising the Union-Find algorithm to solve the edge orientation problem. Several attempts have been made to parallelise that algorithm both on shared memory computers [Cybenko et al., 1988; Anderson and Woll, 1991; Bader and Cong, 2005; Patwary et al.,

Algorithm 9 Edge orientations using the extended Union-Find algorithm

```

1: for all  $c \in \text{local cells}$  do
2:    $e_1, e_2, e'_1, e'_2 \leftarrow$  edges of  $c$ , each consecutive pair sharing a vertex
3:    $o_1 \leftarrow$  permitted relative orientation between  $e_1$  and  $e'_1$ 
4:    $o_2 \leftarrow$  permitted relative orientation between  $e_2$  and  $e'_2$ 
5:    $\text{UNION}(e_1, e'_1, o_1)$ 
6:    $\text{UNION}(e_2, e'_2, o_2)$ 
7: end for
8: for all  $e \in \text{local edges}$  do
9:    $r_e, o_e \leftarrow \text{FIND}(e)$ 
10:   $\text{orientations}[e] \leftarrow o_e$ 
11: end for

```

[2012] and on distributed memory systems [Cybenko et al., 1988; Manne and Patwary, 2010; Harrison et al., 2011; Iverson et al., 2015]. Since Firedrake is meant to scale on today's supercomputers, we are only interested in distributed memory parallelism. Cybenko et al. [1988] attempt both shared memory and distributed memory parallelisation, but their distributed memory algorithm suffered from slowdown with increasing number of processors and constant problem size. Manne and Patwary [2010] propose a distributed memory parallel algorithm, which they demonstrate to scale up to 40 processors for certain graphs. This is, however, far from the scale of supercomputers on which Firedrake is supposed to run. Harrison et al. [2011] achieve good scaling for their purposes, which is connected component labeling for distributed memory visualisation tools. However, the final step of their algorithm uses all-to-all communication to merge the local components, which we expect to become a serious bottleneck for our purposes, since distributed quadrilateral meshes produce numerous local ribbon segments and connections between them. Iverson et al. [2015] compare five parallel algorithms for connected component labeling on distributed memory systems, both theoretically and experimentally. The approach they call *label propagation* is essentially equivalent to the parallelisation of the edge orientation algorithm implemented in Firedrake. Although it was not always the fastest in their evaluations, it did not suffer from the explosive use of memory which affected some other algorithms.

Chapter 4

Two-Stage Form Compiler (TSFC)

We now present the *Two-Stage Form Compiler* (TSFC), a new form compiler incorporated in Firedrake. Its main motivation is to maintain the structure of the input expression as long as possible to facilitate the application of optimisations at the highest possible level of abstraction. Part of achieving this is a novel, structure-preserving method for separating the contributions of a form to the subtensors of the local tensor in discontinuous Galerkin problems (section 4.2). The other core idea is a two-stage approach that cleanly separates the lowering of finite element constructs to tensor algebra in the first stage, from the scheduling of those tensor operations in the second stage. The content of this chapter is derived from Homolya et al. [2017b].

TSFC is essentially another implementation for the *Unified Form Language* (UFL). We also employ the same symbolic geometry pipeline in UFL that UFLACS uses; this enables us to fully support non-affine mesh geometry. TSFC also efficiently traverses complicated expressions, and demonstrates good compile-time performance even for highly complicated forms. As form complexity increases, compile time with TSFC scales similarly as with UFLACS, and much better than earlier FFC representations. TSFC also preserves various structural aspects of the input form longer through the compilation process than either UFLACS or other form compilers.

Section 4.1 introduces the generic view of a compiler that we had in mind when designing TSFC. In section 4.2 we emphasise the importance of preserving structure, and compare our approach with the approach adopted by UFLACS. In section 4.3 we elaborate on the tensor nature of UFL expressions to better understand the tensor structure present in the input form. Sections 4.4 and 4.5 go through the *first stage* of TSFC. Section 4.6 introduces and describes the *intermediate representation*. The *second stage* will be discussed in chapter 7.

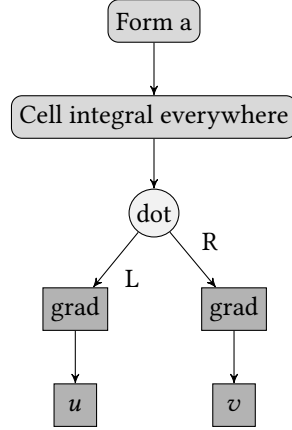


Figure 4.1: Tree representation of the Laplace operator.

4.1 Compilers as series of transformations

A compiler may be understood as a series of expression transformations, called *compiler passes*. A pass takes an expression and produces a new expression, implementing, for example, a simplification or an optimisation, or translating the expression to a different language. A compiler may use several intermediate languages between the source and target language to decompose the translation process into several simpler passes, and especially to facilitate optimisations at the right level of abstraction.

As discussed in section 2.1.2, it is common practice in form compilers to change the integral from physical to reference space, which enables, for example, the precomputation of basis functions at quadrature points. We use this transformation as an example for a compiler pass. Recall that for the Laplace operator

$$\int_K \nabla u \cdot \nabla v \, dx = \int_{\hat{K}} J_K^{-T} \hat{\nabla} \hat{u}_K \cdot J_K^{-T} \hat{\nabla} \hat{v}_K |\det(J_K)| \, d\hat{x} \quad (4.1)$$

where $\hat{u}_K = u \circ F_K$ and similarly $\hat{v}_K = v \circ F_K$. This implies the gradient transforms as $\nabla = J_K^{-T} \hat{\nabla}$, and the integral needs scaling as $dx = |\det(J_K)| \, d\hat{x}$. Fig. 4.1 presents the input UFL form as an expression tree. By locally applying the above substitutions to the arguments, gradients and the integral, we obtain the expression graph in Fig. 4.2. As one can see, J_K^{-T} is referenced twice, and J_K is referenced twice directly, or three times indirectly. A later pass may replace J_K with the evaluation of $\hat{\nabla} F_K$. It is therefore better to think of expressions as *directed acyclic graphs* (DAGs) rather than trees.

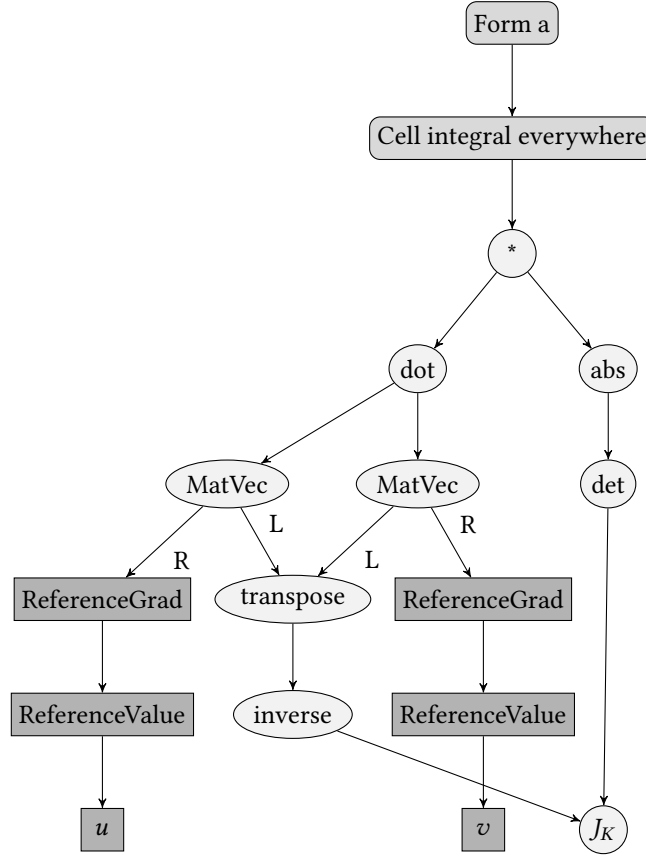


Figure 4.2: The Laplace operator after changing to reference space, equivalent to the right-hand side of eq. (4.1). The presence of two references (arrows) to J_K in place of replicating the J_K node make this a *directed acyclic graph* (DAG) rather than a tree.

4.2 On preserving structure

We mentioned that optimisations are best expressed at a specific level of abstraction, since if the representation is too high-level, the transformation may not be expressible at all, and if the representation is too low-level, difficult analyses may be needed to recognise the optimisation opportunity. It is therefore desirable to postpone representation lowering operations until the appropriate optimisations have been applied. Although UFLACS addresses the most urgent issues of previous form compilers (section 2.2), it is eager to lower high-level constructs early in the form compilation pipeline to keep its internal data structures simple. There are two principal ways in which UFLACS lowers structure early, but TSFC does not: lowering tensor expressions to scalar expressions

and *argument factorisation*.

Argument factorisation rewrites the integrand as a sum of products such that each product contains a term without arguments and for each argument a component of (a derivative of) that argument. This is always possible since the form is multilinear in its arguments. Lowering to scalars is a practical requirement: distributivity rules in vector and tensor algebra are numerous, and completeness of the implementation is necessary for correctness in all cases. For example, the expression $u \times (v \times f)$ has a vector structure preserving argument factorisation:

$$(v \otimes u) \cdot f - (v \cdot u)f$$

where $\otimes$ denotes the outer product of two vectors. The sum-of-product form makes it easy to separate the form into parts that correspond to the subtensors of the local tensor in interior facet integrals (see section 2.1.5). We also note that UFLACS applies argument factorisation irrespective of the integral type. In some cases it turns out to be an effective optimisation as argument-independent expressions are exposed and precomputed outside the inner loops. However, this approach is in general suboptimal. The transformation may require the expansion of products involving one or more arguments which affects an expression in two major ways: *a*) by increasing the operation count (which is usually only partially counteracted by the subsequent loop hoisting); and *b*) by destroying the common subexpressions that naturally arise when a problem is specified in a tensor algebra language.

To demonstrate what UFLACS does, let us consider the following interior facet integral as an example:

$$\int_{\Gamma_{\text{int}}} \llbracket u \rrbracket_n \cdot \{\nabla v\} \, dS \quad (4.2)$$

where $\llbracket u \rrbracket_n$ is the product of the jump of u with the facet normal n , and $\{\nabla v\}$ is the average of ∇v . The equivalent integrand with restrictions is

$$\llbracket u \rrbracket_n \cdot \{\nabla v\} = \frac{1}{2} (u^+ n^+ + u^- n^-) \cdot (\nabla v^+ + \nabla v^-) . \quad (4.3)$$

n is of unit length, and always pointing out of the cell to which it is restricted. UFLACS first lowers the expression to scalars, in this case by expanding the dot product. In a

2-dimensional space, this step yields

$$\begin{aligned} \llbracket u \rrbracket_n \cdot \{\nabla v\} &= \frac{1}{2} \sum_{i=1}^2 (u^+ n_i^+ + u^- n_i^-) ((\nabla v^+)_i + (\nabla v^-)_i) \\ &= \frac{1}{2} \left[(u^+ n_1^+ + u^- n_1^-) ((\nabla v^+)_1 + (\nabla v^-)_1) \right. \\ &\quad \left. + (u^+ n_2^+ + u^- n_2^-) ((\nabla v^+)_2 + (\nabla v^-)_2) \right]. \end{aligned}$$

Argument factorisation is achieved through the application of distributivity rules:

$$\begin{aligned} \llbracket u \rrbracket_n \cdot \{\nabla v\} &= \frac{1}{2} u^+ n_1^+ (\nabla v^+)_1 + \frac{1}{2} u^+ n_1^+ (\nabla v^-)_1 + \frac{1}{2} u^- n_1^- (\nabla v^+)_1 + \frac{1}{2} u^- n_1^- (\nabla v^-)_1 \\ &\quad + \frac{1}{2} u^+ n_2^+ (\nabla v^+)_2 + \frac{1}{2} u^+ n_2^+ (\nabla v^-)_2 + \frac{1}{2} u^- n_2^- (\nabla v^+)_2 + \frac{1}{2} u^- n_2^- (\nabla v^-)_2. \end{aligned}$$

At this point one can easily separate the form based on argument restrictions:

$$(+, +) \Rightarrow \frac{1}{2} u^+ n_1^+ (\nabla v^+)_1 + \frac{1}{2} u^+ n_2^+ (\nabla v^+)_2 \quad (4.4a)$$

$$(+, -) \Rightarrow \frac{1}{2} u^+ n_1^+ (\nabla v^-)_1 + \frac{1}{2} u^+ n_2^+ (\nabla v^-)_2 \quad (4.4b)$$

$$(-, +) \Rightarrow \frac{1}{2} u^- n_1^- (\nabla v^+)_1 + \frac{1}{2} u^- n_2^- (\nabla v^+)_2 \quad (4.4c)$$

$$(-, -) \Rightarrow \frac{1}{2} u^- n_1^- (\nabla v^-)_1 + \frac{1}{2} u^- n_2^- (\nabla v^-)_2. \quad (4.4d)$$

In contrast, TSFC separates the form into parts that correspond to the subtensors without applying argument factorisation. This is achieved by simply exploiting the linearity of the form in its arguments. Since each instance of an argument is either positively or negatively restricted in the form, the following equality holds:

$$\mathcal{F}(v^+, v^-) = \mathcal{F}(v^+, 0) + \mathcal{F}(0, v^-) \quad (4.5)$$

where $\mathcal{F}$ is the form interpreted as a function of two variables, the positively restricted and the negatively restricted argument. Equation (4.5) means that splitting the form based on its contributions to the subtensors of the interior facet tensor is as simple as substituting an appropriately shaped zero for any argument that has the “wrong” restriction. This approach also naturally generalizes to any number of arguments.

Applying this approach to eq. (4.3) directly yields:

$$(+, +) \Rightarrow \frac{1}{2} (u^+ n^+ + 0 n^-) \cdot (\nabla v^+ + \nabla 0) = \frac{1}{2} u^+ n^+ \cdot \nabla v^+ \quad (4.6a)$$

$$(+, -) \Rightarrow \frac{1}{2} (u^+ n^+ + 0 n^-) \cdot (\nabla 0 + \nabla v^-) = \frac{1}{2} u^+ n^+ \cdot \nabla v^- \quad (4.6b)$$

$$(-, +) \Rightarrow \frac{1}{2} (0 n^+ + u^- n^-) \cdot (\nabla v^+ + \nabla 0) = \frac{1}{2} u^- n^- \cdot \nabla v^+ \quad (4.6c)$$

$$(-, -) \Rightarrow \frac{1}{2} (0 n^+ + u^- n^-) \cdot (\nabla 0 + \nabla v^-) = \frac{1}{2} u^- n^- \cdot \nabla v^-. \quad (4.6d)$$

Equations (4.4) and (4.6) are equivalent, but TSFC's approach preserves the tensor structure of the original expression, and does not require rearrangement. In general, the preserved structure can be exploited. For example, COFFEE [Luporini et al., 2015, 2017] explores a *transformation space* with the goal of minimising the operation count of a kernel, and argument factorisation is difficult to undo. This renders inaccessible to COFFEE a part of the transformation space, potentially eliminating the most efficient configurations.

Although we cannot presently provide hard evidence for performance improvement due to preserving the (geometric) tensor structure of the input form, TSFC is designed to have the option of lowering tensor structure sooner or later. Truter [2017] implements optimisations that must precede the lowering of tensor structure, but they can be conveniently done before most other form compiler phases. Section 6.5, however, will show an example where early lowering to scalars leads to poorer description of nonzero patterns, but translating this to measurable performance increase is currently blocked by lack of support for said patterns in global assembly.

Compared to previous form compilers, we essentially applied an inversion of order: lowering of finite element objects and geometric terms happens before either lowering tensor structure or applying argument factorisation. The latter are both optional since argument factorisation does not always improve performance, and explicit loops for tensor operations might be better than unrolled loops. This creates an intermediate representation that is a tensor algebra expression without any finite element objects or geometric terms, as sketched in Fig. 4.3.

4.3 Tensor algebra and index notation in UFL

The tensor nature of UFL expressions is represented as *shape* and *free indices*:

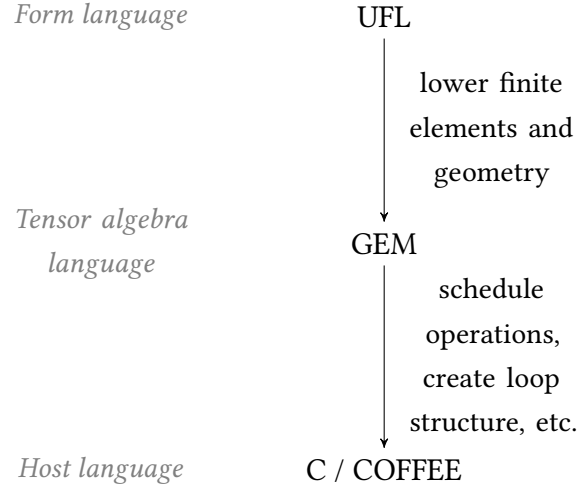


Figure 4.3: TSFC compiles forms in two stages. The first stage translates the evaluation of finite element objects to tensor algebra expressions. The second stage generates C code to evaluate tensor algebra expressions.

shape An ordered list of dimensions and their respective extent, e.g. $(2, 2)$. A dimension is only identified by its position in the shape.

free indices An unordered set of dimensions where each dimension is identified by a symbolic index object. One might think of free indices as an “unrolled shape”.

These traits are an integral part of any UFL expression. For example, let A be a 2×2 matrix, then A has shape $(2, 2)$ and no free indices. $A_{1,1}$ has scalar shape and no free indices; $A_{i,j}$ has scalar shape and free indices i and j ; and $A_{i,1}$ has scalar shape and free index i . UFL has four basic node types related to the tensor nature of its expressions:

- $\text{Indexed}(A, (\alpha_1, \alpha_2, \dots, \alpha_r))$
where A is an expression of rank r , and α is multi-index consisting of fixed indices and free indices. The result has scalar shape, but all the free indices of α become free indices of the result, i.e. Indexed can convert shape to free indices.
- $\text{IndexSum}(e, i)$
Tensor contraction of the expression e over the free index i .
- $\text{ComponentTensor}(e, \alpha)$
where $\alpha := (\alpha_1, \alpha_2, \dots, \alpha_k)$ is a multi-index consisting of free indices, and e is an

expression with free indices $\alpha_1, \alpha_2, \dots, \alpha_k$ (at least). These free indices are turned into shape, in the order as they appear in α .

- `ListTensor( $e_1, e_2, \dots, e_n$ )`
where $e_1, e_2, \dots, e_n$ are expressions of shape $(s_1, s_2, \dots, s_r)$. The result is a concatenation of (the otherwise unrelated) expressions $e_1, e_2, \dots, e_n$, with shape $(n, s_1, s_2, \dots, s_r)$.

The index notation provides a low-level representation for many vector and matrix operations. For example, dot product, element-wise multiplication and outer product are all expressible using only multiplication of scalars and some of the above tensor operations:

- Dot product: $S = \sum_i u_i v_i$
`dot(u, v)  $\Rightarrow$  IndexSum(u[i] * v[i], i)`
- Element-wise multiplication: $w_i = u_i v_i \quad \forall i$
`elem_mult(u, v)  $\Rightarrow$  ComponentTensor(u[i] * v[i], (i,))`
- Outer product: $A_{ij} = u_i v_j \quad \forall i, j$
`outer(u, v)  $\Rightarrow$  ComponentTensor(u[i] * v[j], (i, j))`

4.4 UFL preprocessing and modified terminals

TSFC relies on UFL to preprocess the form. The most important step in this procedure is to change the integral from physical to reference space (see sections 2.1.2 and 4.1). After this transformation in eq. (4.1), UFL does various simplifications on the form: the Jacobian is replaced with the gradient of the spatial coordinates, determinants are expanded, divergence and curl are expressed with tensor algebra on gradients, and various products are expanded using the index notation. Section 4.3 demonstrated that lowering to the index notation introduces `ComponentTensor` nodes; however, to make this discussion easier to follow, we simplify most of them away, even though in reality they are only eliminated at a later stage (section 7.1). Finally, gradients and restrictions are propagated to the terminals. This reduces the number of language constructs that the form compiler needs to handle while preserving a rich language for the user. As a result of these operations, eq. (4.1) becomes:

$$\begin{aligned}
d &:= \left(\frac{\partial x}{\partial \hat{x}} \right)_{1,1} \left(\frac{\partial x}{\partial \hat{x}} \right)_{2,2} - \left(\frac{\partial x}{\partial \hat{x}} \right)_{1,2} \left(\frac{\partial x}{\partial \hat{x}} \right)_{2,1} &> \det(J) \\
B &:= \begin{bmatrix} \left(\frac{\partial x}{\partial \hat{x}} \right)_{2,2} & - \left(\frac{\partial x}{\partial \hat{x}} \right)_{1,2} \\ - \left(\frac{\partial x}{\partial \hat{x}} \right)_{2,1} & \left(\frac{\partial x}{\partial \hat{x}} \right)_{1,1} \end{bmatrix} &> \text{adj}(J) \\
K &:= \left[\frac{B_{i_1, i_2}}{d} \right]_{i_1, i_2} &> \text{inv}(J) \\
\int \sum_{i_5} \left(\sum_{i_3} K_{i_3, i_5} (\hat{\nabla} \hat{u}_K)_{i_3} \right) \left(\sum_{i_4} K_{i_4, i_5} (\hat{\nabla} \hat{v}_K)_{i_4} \right) |d| \, d\hat{x}
\end{aligned} \tag{4.7}$$

where $]e[_\alpha$ is a notation for $\text{ComponentTensor}(e, \alpha)$. This denotes a single expression, although some subexpressions are assigned names for simpler presentation.

A *terminal* is a node in an expression graph with no children. UFL terminals are: *a)* constant literals, *b)* coefficients, *c)* arguments, *d)* geometric quantities, and *e)* quadrature weight. Various *terminal modifiers* may wrap a terminal. TSFC only needs to handle the following terminal modifiers:

- **PositiveRestricted, NegativeRestricted**
Restriction to one particular side of the facet in an interior facet integral.
- **ReferenceGrad**
Gradient in the reference space, possibly applied multiple times.

A *modified terminal* is a terminal with a set of modifiers attached. In this set-up, the translation of modified terminals is independent of their context (surrounding outer expression), and the translation of the overall expression does not need to know more about the modified terminals than their shape and free indices.

4.5 Substitution of modified terminals

The substitution rules for modified terminals mimic the evaluation rules of FFC:

- **Arguments.** FLAT [Kirby, 2004] provides the tabulation matrix that contains the value of each basis function at each quadrature point for the finite element of the argument. This becomes a compile-time constant matrix in the intermediate representation. This matrix is then indexed with the quadrature index and the corresponding argument index.

- **Coefficients.** We need a tensor algebra expression that evaluates the coefficient at each quadrature point. The matrix-vector product of the element tabulation matrix with the local basis function coefficients gives exactly that. That vector for the cell is given as a kernel argument, which is represented as a run-time value in the intermediate representation.
- **Geometric quantities.** Because Firedrake represents the mesh coordinates as a finite element field, the Jacobian and further derivatives of the coordinate field are simply evaluated as coefficients. Other geometric terms typically appear in facet integrals (e.g., facet normal), and, as a consequence of UFL preprocessing, are expressed in reference space. They depend on the cell type, which is known at compile-time, and often on the facet, which is provided as a kernel argument at run-time. Thus the evaluation of a geometric term usually involves indexing into a compile-time constant tensor with the facet index.

Applying these evaluation rules, the modified terminals of eq. (4.7) are replaced with the following expressions:

$$\hat{\mathbf{V}}\hat{v}_K \rightarrow \begin{bmatrix} E_{q,j}^{(1)} & E_{q,j}^{(2)} \end{bmatrix} \quad (4.8)$$

$$\hat{\mathbf{V}}\hat{u}_K \rightarrow \begin{bmatrix} E_{q,k}^{(1)} & E_{q,k}^{(2)} \end{bmatrix} \quad (4.9)$$

$$\frac{\partial x}{\partial \hat{x}} \rightarrow \begin{bmatrix} \sum_r C_{q,r}^{(1,1)} c_r & \sum_r C_{q,r}^{(1,2)} c_r \\ \sum_r C_{q,r}^{(2,1)} c_r & \sum_r C_{q,r}^{(2,2)} c_r \end{bmatrix} \quad (4.10)$$

where $E^{(b)}$ is the tabulation matrix of the first derivative of the finite element of u and v in the b -th direction, $C^{(a,b)}$ is the tabulation matrix of the first derivative in the b -th direction of the a -th component of the coordinate element, and c is the vector of the local basis function coefficients of the coordinates. This general treatment of the Jacobian also works for non-simplex cells and higher-order geometries. We will discuss optimisations for affine simplices in section 4.7. Since indexing a tensor with fixed indices immediately selects the appropriate entry, these substitutions yield the following integrand expression:

$$\begin{aligned}
J^{(1,1)} &:= \sum_r C_{q,r}^{(1,1)} c_r & d &:= J^{(1,1)} J^{(2,2)} - J^{(1,2)} J^{(2,1)} \\
J^{(1,2)} &:= \sum_r C_{q,r}^{(1,2)} c_r & B &:= \begin{bmatrix} J^{(2,2)} & -J^{(1,2)} \\ -J^{(2,1)} & J^{(1,1)} \end{bmatrix} \\
J^{(2,1)} &:= \sum_r C_{q,r}^{(2,1)} c_r & K &:= \left[\frac{B_{i_1, i_2}}{d} \right]_{i_1, i_2} \\
J^{(2,2)} &:= \sum_r C_{q,r}^{(2,2)} c_r
\end{aligned}$$

$$\int \sum_{i_5} \left(\sum_{i_3} K_{i_3, i_5} \begin{bmatrix} E_{q,k}^{(1)} & E_{q,k}^{(2)} \end{bmatrix}_{i_3} \right) \left(\sum_{i_4} K_{i_4, i_5} \begin{bmatrix} E_{q,j}^{(1)} & E_{q,j}^{(2)} \end{bmatrix}_{i_4} \right) |d| \, d\hat{x}.$$

Aggregating the integrand values for each quadrature point with quadrature weight w_q gives the intermediate representation as

$$\sum_q w_q |d| \sum_{i_5} \left(\sum_{i_3} K_{i_3, i_5} \begin{bmatrix} E_{q,k}^{(1)} & E_{q,k}^{(2)} \end{bmatrix}_{i_3} \right) \left(\sum_{i_4} K_{i_4, i_5} \begin{bmatrix} E_{q,j}^{(1)} & E_{q,j}^{(2)} \end{bmatrix}_{i_4} \right). \quad (4.11)$$

This representation only contains tensor algebra operators applied to compile-time constant tensors and a run-time argument to the local kernel. All the finite element and geometric terms have been lowered, the first stage of form compilation is complete.

4.6 The intermediate representation: GEM

Equation (4.11) is represented in GEM as anticipated in Fig. 4.3. GEM is a tensor algebra language very similar to UFL in its tensor algebra features, but it contains nothing specific to the finite element method. This is satisfactory since the evaluation of finite element objects and geometric terms is expressible using just tensor algebra. On the other hand, since GEM mimics the tensor algebra features of UFL and lacks other features, one may rightly ask why the tensor algebra subset of UFL could not be adopted as the intermediate representation. The reason for the creation of GEM is to relax some of the constraints on *free indices* which are intrinsic in UFL.

Free indices are just an “unrolled shape” in UFL, that may be “rolled back” any time. For any operation, free indices of the operands are subject to the same constraints that apply to shape. For example, tensors must have equal shape for addition, consequently, they also must have the same free indices. GEM interprets free indices differently. A

free index in GEM simply means that the value of the expression depends on that index. For example, let u be a 3-dimensional vector (no free indices). u_i and u_j are scalars with free indices i and j , respectively. So far, it is identical to UFL. However, in GEM $u_i + u_j$ is just an expression that depends on both i and j , that is a scalar with free indices i and j , while in UFL the same expression is illegal because the free indices do not match: the addition of a 3-dimensional vector with itself cannot produce a 3×3 matrix.

While the enforcement of strict shape checking is valuable for preventing user errors in a weak form, this strictness becomes an obstacle when the modified terminals are replaced with their respective evaluation rules. For example, we replaced $\hat{V}\hat{v}_K$ with $\begin{bmatrix} E_{q,j}^{(1)} & E_{q,j}^{(2)} \end{bmatrix}$ in section 4.5. In UFL this would mean replacing a vector with a third-order tensor, which yields an illegal expression in the general case. In GEM it just replaces a vector with another vector which depends on q (quadrature index) and j (argument index). There is eventually a summation over q , and the dependence on j gives the components of the local tensor. This difference in free indices poses no difficulty for the UFL $\rightarrow$ GEM translation since GEM is strictly more permissive.

Overview of GEM GEM was intentionally designed to be very similar to UFL in its tensor algebra features. In common with UFL, the tensor nature of expressions can be expressed as *shape* or *free indices* (see section 4.3). Here we give a list of all GEM nodes:

- **Terminals:**

- Literal (tensor literal)
- Zero (all-zero tensor)
- Identity (identity matrix)
- Variable (run-time value, for kernel arguments)

- **Scalar operations:**

- *Binary operators*: Sum, Product, Division, Power, MinValue, MaxValue
- *Unary operators*: MathFunction (e.g. sin, cos)
- Comparison ($>$, $\geq$, $=$, $\neq$, $<$, $\leq$): compares numbers, returns Boolean
- *Logical operators*: LogicalAnd, LogicalOr, LogicalNot
- Conditional: selects between a “true” and a “false” expression based on a Boolean valued *condition*

- **Index types:**
 - `int` (fixed index)
 - `Index` (free index): creates a loop at code generation
 - `VariableIndex` (unknown fixed index): index value only known at run-time, e.g. facet number
- **Tensor nodes:** `Indexed`, `ComponentTensor`, `IndexSum`, `ListTensor`
See section 4.3 for details.
- **Special nodes:**
 - `Delta` (Kronecker delta)
 - `Concatenate`: vectorises each operand and concatenates them.

Most supported operations, such as addition and multiplication, naturally require scalar shape. So operands with non-scalar shape need indexing before most operations, but the shape can be restored by wrapping the result in a `ComponentTensor`.

4.7 Basic optimisations

We finally discuss some key optimisations applied during the first stage of TSFC.

4.7.1 Constant folding for scalars

Constant folding means replacing computations whose all operands are known at compile time with the result of the computation. This is a standard technique in modern optimising compilers. TSFC implements a limited version of it at the GEM level: only computations involving *scalars* are folded. An especially important case of this is multiplication by zero: $0 \cdot X = 0$, which can eliminate an arbitrarily complex expression X . Zero tensors can arise, for example, as tabulation matrices for higher derivatives than the degree of the finite element.

4.7.2 Cellwise constant quantities

We saw in section 4.5 that the evaluation of arguments and coefficients involves a tabulation matrix $M_{q,r}$ that contains the value of each basis function at each quadrature

point. When an argument or a coefficient, or more commonly a derivative of it, is cellwise constant, then evaluation need not happen for every quadrature point. So instead of the matrix $M_{q,r}$ we create a vector M_r which does not have the quadrature index as a free index. This change in combination with *loop hoisting*, which we will discuss in section 7.2, hoists the evaluation of all cellwise constant subexpressions out of the quadrature loop.

4.7.3 Fast Jacobian evaluation for affine cells

The Jacobian of the coordinate transformation appears in virtually every form after preprocessing, even the simplest ones. In those cases an inefficient Jacobian evaluation may likely become the dominant cost of assembly. Consequently, FEniCS has, and Firedrake used to have, efficient hand-written code snippets for Jacobian evaluation on affine simplices. As described in section 4.5, we now implement Jacobian evaluation as coefficient evaluation. This allows us to handle arbitrary order geometries with a single code path, but it also implies that each entry of the Jacobian is evaluated as a matrix-vector product. For affine cells, the Jacobian is cellwise constant, so this reduces to a dot product of two vectors.

The code snippets evaluate each Jacobian entry with a single subtraction, which is possible because of the choice of reference simplices and the properties of first-order Lagrange coordinates. For example, the vertices of the reference tetrahedron are $(0, 0, 0)$, $(0, 0, 1)$, $(0, 1, 0)$, and $(1, 0, 0)$. Consequently, the gradients of the basis functions are $(-1, -1, -1)$, $(0, 0, 1)$, $(0, 1, 0)$, and $(1, 0, 0)$, which form the matrix

$$\begin{bmatrix} -1 & 0 & 0 & 1 \\ -1 & 0 & 1 & 0 \\ -1 & 1 & 0 & 0 \end{bmatrix}.$$

Each row corresponds to the *tabulation vector* of a particular partial derivative of the scalar Lagrange element. From here it is easy to see that each Jacobian entry can be evaluated as a subtraction between two degrees of freedom of the coordinate field. This is also true for triangles or indeed for any simplex as long as the reference simplex is chosen analogously. Therefore, we introduced another optimisation rule which applies to all coefficient evaluation: if the quantity is cellwise constant and the tabulation vector has at most two nonzero entries, then we expand the dot product that evaluates the coefficient. This, of course, composes with constant folding, so the multiplications with

zeros vanish, and we obtain a formulation that is performance equivalent to the code snippets.

Chapter 5

FInAT: exposing structure in finite elements

We have shown in section 2.1.4 that certain optimisations rely on the structure inherent to *some* finite elements, and that the existing interface between the form compiler and FIAT, the finite element library, is inadequate to communicate this structure. Therefore, we introduce a more powerful interface between the form compiler and the library providing the finite elements, along with a more abstract, smarter library of finite elements called FInAT that explicitly communicates the structure of elements. The content of this chapter is derived from [Homolya, Kirby, and Ham \[2017a\]](#).

Unlike FIAT, “FInAT Is not A Tabulator.” Instead, it provides symbolic expressions for the evaluation of finite element basis functions at a set of points, henceforth called *tabulation expressions*. Thus FInAT is able to express the structure that is intrinsic to some finite elements. To facilitate seamless integration with TSFC, tabulations are provided in the tensor algebra language GEM, the intermediate representation of TSFC described in section 4.6, so TSFC can directly insert tabulation expressions into partially compiled form expressions. Since TSFC originally used FIAT for an implementation of finite elements [[Homolya et al., 2017b](#)], changes were made to the first stage to use FInAT instead.

The rest of this chapter describes the structure-exposing element implementations in FInAT. Combined, they enable the construction of continuous, discontinuous, $H(\text{div})$ and $H(\text{curl})$ conforming tensor product elements, exposing enough structure to achieve optimal algorithmic complexity for high-order finite element assembly. Chapter 6 will give the form compiler algorithms implemented in TSFC that exploit this structure.

5.1 Generic FIAT element wrapper

Let $\{\xi_q\}_{q=1}^{N_q}$ be a set of (quadrature) points on the reference cell, and $\{\Psi_i\}_{i=1}^{N_f}$ be the basis functions of a reference finite element. If FIAT implements this element, then it can produce a tabulation matrix Ψ such that $\Psi_{iq} := \Psi_i(\xi_q)$. Since GEM can represent literal matrices and indexing, FInAT can invoke FIAT and construct a trivial tabulation expression, providing the substitution:

$$\Psi_i(\xi_q) \mapsto \Psi_{iq} \quad (5.1)$$

and similarly for derivatives of basis functions. This generic wrapper removes the need for the form compiler to interface both FIAT and FInAT at the same time, while all FIAT elements continue to be available with no regression.

5.2 Vector and tensor elements

A *vector element* constructs a vector-valued element by duplicating a scalar-valued element for each component. Fluid velocities, e.g., are often represented using vector elements of Lagrange elements. Let $\{\Psi_\alpha^*\}_{\alpha \in A}$ be the basis of a scalar-valued finite element. When the basis has a tensor-product structure, $\alpha = (\alpha_1, \alpha_2, \dots, \alpha_s)$ is a multi-index. The corresponding d -dimensional vector element has basis

$$\Psi_{(\alpha,j)} = \Psi_\alpha^* \hat{e}_j \quad (5.2)$$

where $\hat{e}_j$ is the d -dimensional unit vector whose j -th coordinate is 1.

The scalar element may have its own structure, so let $\mathcal{E}\{\Psi_\alpha^*(\xi_q)\}$ denote its tabulation expression at points $\{\xi_q\}_{q=1}^Q$. Then FInAT constructs the following symbolic expression for the evaluation of the k -th component of the vector element:

$$\Psi_{(\alpha,j)k}(\xi_q) \mapsto \mathcal{E}\{\Psi_\alpha^*(\xi_q)\} \delta_{jk}. \quad (5.3)$$

A vector element is a rank-1 *tensor element*.¹ FInAT can also evaluate rank- n tensor elements as

$$\Psi_{(\alpha,\nu)\kappa}(\xi_q) \mapsto \mathcal{E}\{\Psi_\alpha^*(\xi_q)\} \prod_{i=1}^n \delta_{\nu_i \kappa_i}. \quad (5.4)$$

¹Note the distinction between “tensor element” and “tensor product” element. By *tensor element* we mean a finite element whose function space members take on tensorial values. By *tensor product element* we mean that the basis functions can be written as a product of lower-dimensional basis functions.

5.3 Tensor product element

Here we follow [McRae et al. \[2016\]](#) for a definition of the *tensor product* element¹. Let $K_1 \subset \mathbb{R}^{d_1}$ and $K_2 \subset \mathbb{R}^{d_2}$ be reference cells, the reference tensor product cell $K_1 \times K_2$ is defined as

$$K_1 \times K_2 = \{(\hat{x}_1, \dots, \hat{x}_{d_1+d_2}) \in \mathbb{R}^{d_1+d_2} \mid (\hat{x}_1, \dots, \hat{x}_{d_1}) \in K_1, (\hat{x}_{d_1+1}, \dots, \hat{x}_{d_1+d_2}) \in K_2\}. \quad (5.5)$$

Similarly, let $\{\xi_{q_1}^{(1)}\}_{q_1=1}^{Q_1}$ and $\{\xi_{q_2}^{(2)}\}_{q_2=1}^{Q_2}$ be quadrature points on K_1 and K_2 respectively. The tensor product point set $\{\xi_{(q_1, q_2)}\}_{(q_1, q_2)}$ is defined as

$$\xi_{(q_1, q_2)} = (\hat{x}_1, \dots, \hat{x}_{d_1}, \hat{x}_{d_1+1}, \dots, \hat{x}_{d_1+d_2}) \quad (5.6)$$

$$\text{where } (\hat{x}_1, \dots, \hat{x}_{d_1}) = \xi_{q_1}^{(1)}, \text{ and} \quad (5.7)$$

$$(\hat{x}_{d_1+1}, \dots, \hat{x}_{d_1+d_2}) = \xi_{q_2}^{(2)}.$$

Let $\{\Psi_{i_1}^{(1)}\}_{i_1=1}^{N_{f,1}}$ and $\{\Psi_{i_2}^{(2)}\}_{i_2=1}^{N_{f,2}}$ be finite element bases on the reference cells K_1 and K_2 respectively. The tensor product element on reference cell $K := K_1 \times K_2$ has basis functions

$$\Psi_{(i_1, i_2)}(\hat{x}_1, \dots, \hat{x}_{d_1+d_2}) = \Psi_{i_1}^{(1)}(\hat{x}_1, \dots, \hat{x}_{d_1}) \Psi_{i_2}^{(2)}(\hat{x}_{d_1+1}, \dots, \hat{x}_{d_1+d_2}). \quad (5.8)$$

When both the finite element and the point set are tensor products of finite elements and point sets with matching dimensions, as in Fig. 5.1, FInAT constructs the tabulation expression as the tensor product of the tabulations of factor elements:

$$\Psi_{(i_1, i_2)}(\xi_{(q_1, q_2)}) \mapsto \mathcal{E} \left\{ \Psi_{i_1}^{(1)}(\xi_{q_1}^{(1)}) \right\} \mathcal{E} \left\{ \Psi_{i_2}^{(2)}(\xi_{q_2}^{(2)}) \right\}. \quad (5.9)$$

If the factor elements have no further structure, this may simply mean $\Psi_{i_1 q_1}^{(1)} \Psi_{i_2 q_2}^{(2)}$, where $\Psi^{(1)}$ and $\Psi^{(2)}$ are FIAT-provided tabulation matrices.

5.4 Collocated quadrature points

Spectral elements [\[Karniadakis and Sherwin, 2013; Patera, 1984\]](#) use Lagrange polynomials whose nodes are collocated with quadrature points. Typically, Gauss–Lobatto–Legendre (GLL) quadrature points are chosen for (continuous) interval elements, while Gauss–Legendre (GL) quadrature points are better for discontinuous elements. When

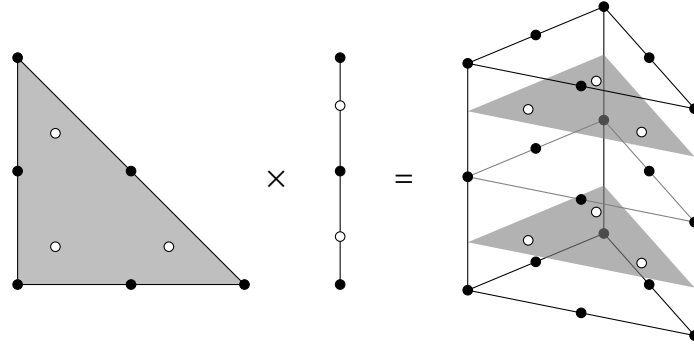


Figure 5.1: Second order triangular prism element and quadrature points. Black balls denote point evaluation nodes, and white balls mark quadrature points. The element and the quadrature rule are both tensor products of triangular and interval elements and quadrature rules respectively. Consequently, the tabulation matrix of the prism element is the Kronecker product of the tabulation matrices of the triangular and interval elements.

these nodes match the quadrature rule that is used to approximate the integral, then the tabulation matrix becomes the identity matrix. In other words, FInAT can return

$$\Psi_i(\xi_q) \mapsto \delta_{iq}, \quad (5.10)$$

and then the form compiler can optimise this away with a resulting simplification of the loop nests.

Another advantage of spectral elements is that they result in a better condition number of the assembled linear system than equidistant Lagrange elements, especially for higher polynomial degrees. We have added GaussLobattoLegendre and GaussLegendre elements to FIAT, and the appropriate element wrappers in FInAT. These wrappers are similar to the generic wrapper in section 5.1, except that they symbolically recognise if the tabulation points match the element nodes, in which case they replace the zeroth derivative according to eq. (5.10). GLL and GL are defined as one-dimensional quadrature rules, and they do not directly generalise to simplices. However, using tensor product elements one can construct higher-dimensional equivalents on box cells such as quadrilaterals and hexahedra.

5.5 Enriched element

By *enriched element* we mean the direct sum of two (or more) finite elements. A basis for such an element is given by the concatenation of the bases of the summands. Let V_1 and V_2 be finite elements on the reference cell K , and let $\{\Psi_{i_1}^{(1)}\}_{i_1=1}^{N_{f,1}}$ and $\{\Psi_{i_2}^{(2)}\}_{i_2=1}^{N_{f,2}}$ be their bases respectively. The enriched element $V := V_1 \oplus V_2$ has basis $\{\Psi_i\}_{i=1}^{N_{f,1}+N_{f,2}}$ such that

$$\Psi_i = \begin{cases} \Psi_i^{(1)} & \text{when } 1 \leq i \leq N_{f,1}, \\ \Psi_{i-N_{f,1}}^{(2)} & \text{when } N_{f,1} < i \leq N_{f,1} + N_{f,2}. \end{cases} \quad (5.11)$$

For example, the triangular Mini element $P_2 \oplus B_3$ denotes the space of quadratic polynomials enriched by a cubic “bubble” function [Arnold et al., 1984].

To implement this element, we utilise the newly introduced Concatenate node of GEM, which flattens the shape of its operands and concatenates them. For example, let a be a 2×2 matrix, b a scalar, and c a vector of length 3. Then $\text{Concatenate}(a, b, c)$ is a vector of length 8 such that its first four entries correspond to the entries of a , its fifth entry to b , and its last three entries to c . We use this node to concatenate the basis functions of the subelements.

Let $\mathcal{E}_{i_1 q}^{(1)}$ and $\mathcal{E}_{i_2 q}^{(2)}$ denote the tabulation expressions for subelements V_1 and V_2 . To correctly use Concatenate, basis function indices must become *shape* first, so the tabulation of V is

$$\text{Concatenate} \left(\left[\mathcal{E}_{i_1 q}^{(1)} \right]_{i_1}, \left[\mathcal{E}_{i_2 q}^{(2)} \right]_{i_2} \right)_i. \quad (5.12)$$

In the general case, any of q , i_1 , and i_2 can be a multi-index. Moreover, enriched elements are not limited to scalar-valued finite elements, however, all subelements must have the same value shape. With value multi-index κ , eq. (5.12) generalises to

$$\text{Concatenate} \left(\left[\mathcal{E}_{i_1 q \kappa}^{(1)} \right]_{i_1}, \left[\mathcal{E}_{i_2 q \kappa}^{(2)} \right]_{i_2} \right)_i. \quad (5.13)$$

Note that q and κ just pass through as free multi-indices, and only the basis function multi-indices i_1 and i_2 are replaced by the unified index i . We later show how enriched elements are destructured in the form compiler to recover the structure within the subelements.

5.6 Value modifier element wrappers

The facilities of FInAT for tensor product and enriched elements allow us to implement $H(\text{div})$ and $H(\text{curl})$ elements in a structure-revealing way. These elements are important

in stable mixed finite element discretisations. Firedrake supports the whole Q^- family of the *Periodic Table of the Finite Elements* [Arnold and Logg, 2014]. This family is defined on cube cells, and its members can be constructed out of interval elements. The continuous Q and the discontinuous dQ elements are *just* tensor products of interval elements, while the construction of $H(\text{div})$ and $H(\text{curl})$ conforming elements is slightly more complicated. McRae et al. [2016] provide constructions for $H(\text{div})$ and $H(\text{curl})$ elements. For example quadrilateral Raviart–Thomas [1977] (RTCF) elements can be constructed as

$$\text{RTCF}_n = \text{HDiv}(\mathbf{P}_n \otimes \mathbf{dP}_{n-1}) \oplus \text{HDiv}(\mathbf{dP}_{n-1} \otimes \mathbf{P}_n) \quad (5.14)$$

where $\mathbf{P}_n$ and $\mathbf{dP}_n$ denote n -th order continuous and discontinuous Lagrange elements (on intervals), $\otimes$ denotes tensor product, $\oplus$ implies an enriched element, and HDiv is a special element wrapper that applies a transformation to the values of the basis functions of a tensor product element. Concretely, if $\Psi_i(\hat{x})$ is a basis function of $\mathbf{P}_n \otimes \mathbf{dP}_{n-1}$, then the corresponding basis function of $\text{HDiv}(\mathbf{P}_n \otimes \mathbf{dP}_{n-1})$ is

$$\Psi_i^*(\hat{x}) = \begin{bmatrix} -\Psi_i(\hat{x}) \\ 0 \end{bmatrix}. \quad (5.15)$$

Note that Ψ_i is a scalar-valued function, while Ψ_i^* is a vector field. If HDiv is applied to $\mathbf{dP}_{n-1} \otimes \mathbf{P}_n$, then

$$\Psi_i^*(\hat{x}) = \begin{bmatrix} 0 \\ \Psi_i(\hat{x}) \end{bmatrix}. \quad (5.16)$$

This difference may seem surprising, but it is possible because HDiv is defined as a long switch-case: based on the continuity and occasionally the reference value transformation of each factor element, it applies the transformation that makes the result suitable for building an $H(\text{div})$ conforming element. HCurl behaves similarly for building $H(\text{curl})$ conforming elements.

We have listed only two cases of HDiv in eqs. (5.15) and (5.16). For all cases of HDiv and HCurl , as well as for a complete description of the construction of Q^- family elements, we refer to McRae et al. [2016]. What is important at the moment, is that these element wrappers

- do not change or destroy the structure of the element they are applied to. (In the general case, i is a multi-index.)
- allow us to build the $H(\text{div})$ and $H(\text{curl})$ conforming elements of the Q^- family, together with tensor product and enriched elements.

Chapter 6

Automatically exploiting the structure in finite elements

Expressing the inherent structure of finite elements is necessary, but not sufficient for achieving optimal code generation. The other main ingredient is algorithms for rearranging *tensor contractions* (called IndexSum in UFL and GEM) such that an optimal assembly algorithm is achieved. We have implemented these algorithms in TSFC as GEM-to-GEM transformers.

We first review related work in section 6.1, derived from Homolya et al. [2017b, §4.5]. The remainder of this chapter then discusses the form compiler algorithms implemented in TSFC. The content of that discussion is derived from Homolya et al. [2017a].

6.1 Related work

Since efficient implementation of tensor algebra expressions is not specific to the finite element method, the related work is broad in scope. Various tensor algebra libraries and domain-specific languages [Baumgartner et al., 2005; Epifanovsky et al., 2013; Sanders et al., 2010; Solomonik et al., 2013] have been developed for quantum chemistry computations. However, while the *working set size*¹ of a finite element local assembly kernel rarely exceeds a megabyte, the tensors in quantum chemistry computations are typically huge, even reaching petabytes in size. Consequently, the optimised implementation of individual tensor operations on distributed-memory parallel computers is usually the major concern, as is the case with SIAL [Sanders et al., 2010], libtensor

¹The amount of memory required to carry out a sequence of operations.

[Epifanovsky et al., 2013], and CTF [Solomonik et al., 2013]. Similarly, TensorFlow [Abadi et al., 2016] serves to instrument tensor computations for machine learning purposes on large-scale heterogeneous computing systems, so most of their challenges do not overlap with ours. The *Tensor Contraction Engine* (TCE) [Baumgartner et al., 2005], however, also considers optimisations on the whole tensor algebra expression, which are applicable to our case as well. Some of these ideas were also applied to do efficient tensor contractions on GPUs [Nelson et al., 2015].

Algebraic transformations on tensor algebra expressions can produce equivalent formulations that require many fewer operations to compute. Lam et al. [1996, 1997] first studied the problem of *single-term optimisation*, also known as *strength reduction*: the operation-minimal evaluation of tensor contractions over the product of several tensors with various indices. They prove that the problem is NP-complete, but they also provide a pruning search strategy that is efficient in practice, since a single term typically consists of only a limited number of tensors. Kschischang et al. [2001] show that such a single term can be characterised by a *factor graph*. If the factor graph is *cycle-free* then it straightforwardly yields the unique optimal factorisation. Otherwise different factorisations can be produced by various transformations that break the cycles. Considering the sum of single terms, Hartono et al. [2005] achieved further improvements over single-term optimisation by considering various factorisations of the whole expression. Since the space of different factorisations is huge, they relied on various search techniques. Exploiting *common subexpression elimination* during the search resulted in further gains [Hartono et al., 2006].

Although rearranging compound expressions may drastically reduce the required computational work in some cases, the efficient implementation of primitive tensor operations is necessary for the effective use of hardware. Traditionally, BLAS (Basic Linear Algebra Subroutines) implementations provide optimised dense linear algebra operations; Springer and Bientinesi [2016] present efficient strategies for generalising to tensor-tensor multiplication. However, these routines are optimised for large tensors and matrices. Eigen [Guennebaud et al., 2010] is a templated, header-only C++ library that provides efficient linear algebra routines for small tensors. M  thing et al. [2017] use libxsmm [Heinecke et al., 2016] in their high-order discontinuous Galerkin code, which contains heavily optimised BLAS-like routines for small matrices. Kjolstad et al. [2017] present the *Tensor Algebra Compiler* (taco), which generates efficient code for primitive tensor operations, especially for various sparse array formats. While finite element local assembly mostly involves dense tensor operations, Kirby [2017] describes

a theory of reference transformation that requires multiplications with sparse matrices for finite elements with stronger than C^0 continuity.

Since tensors in quantum chemistry computations are huge, optimising the memory requirement is of great importance so that computations can fit in the main memory. While this is not an issue for finite element local assembly, fitting the working set into the cache can have an important performance impact. The evaluation of tensor expressions in a scalar programming language such as C requires the introduction of *array temporaries* which may be indexed. The order in which these temporaries are produced and consumed affects the working set size of the kernel. Lam et al. [1999a, 2011] give an algorithm for the evaluation order of nodes of an expression tree that minimises the maximal total size of live temporaries. Their work can be understood as a generalisation of the Sethi-Ullman algorithm [1970] where nodes are not assumed to have the same size, instead each node can have an arbitrary size. Since tensor valued temporaries are both produced and consumed by loops, *loop fusion* can eliminate dimensions of these temporaries, thus reducing the required memory, as we will demonstrate in section 7.2. Lam et al. [1999b] formalise the legality of loop fusions over expression trees and their effect on the memory requirement. They give an algorithm for finding the memory-optimal set of loop fusions based on dynamic programming and pruning, with an exponential complexity in the number of index variables. Allam et al. [2006] propose an alternative algorithm that reduces this problem to an *integer linear programming* formulation. Cociorva et al. [2002] studied space-time trade-off optimisation in this context, by allowing some recomputations for further loop fusion opportunities. Since memory use optimisation is tightly coupled with code generation, we will get back to these aspects in chapter 7.

6.2 Delta cancellation: simple case

One important application of *delta cancellation*, by which we mean the simplification of loop nests involving Kronecker deltas, is found in our handling of mesh coordinates. The coordinate field is often stored in numerical software by assigning coordinates to each vertex of the mesh. This is equivalent to a vector- P_1 or vector- Q_1 finite element space. In fact, Firedrake represents coordinates as an ordinary finite element field, which facilitates support for higher-order geometries.

Since integral scaling involves the Jacobian of the coordinate transformation, virtually every finite element kernel is parametrised by this vector field. We have seen in

section 5.2 that the FInAT implementation of vector elements contains Kronecker delta nodes; we now show how to simplify them away.

Consider a single entry J_{ab} of the Jacobian matrix. We need its evaluation at each quadrature point ξ_q , let us call this $j_q := J_{ab}(\xi_q)$. Let $\{\Phi_i\}_{i=1}^{N_c}$ be the basis functions of the coordinate element, and let $\{c_i\}_{i=1}^{N_c}$ denote the local coefficients of this basis, so

$$j_q := J_{ab}(\xi_q) = \sum_{i=1}^{N_c} c_i \left[\frac{\partial \Phi_i}{\partial \hat{x}_b}(\xi_q) \right]_a. \quad (6.1)$$

Suppose we have a vector- P_2 coordinate element, then FInAT gives:

$$\left[\frac{\partial \Phi_i}{\partial \hat{x}_b}(\xi_q) \right]_a \mapsto \Phi_{i_1 q}^{(b,*)} \delta_{i_2 a} \quad (6.2)$$

where $i = (i_1, i_2)$ and $\Phi^{(b,*)}$ is the tabulation expression of the b -th derivative of the scalar P_2 element. Applying the substitution yields:

$$j_q = \sum_{i_1, i_2} c_{(i_1, i_2)} \Phi_{i_1 q}^{(b,*)} \delta_{i_2 a}. \quad (6.3)$$

Then the tensor product is disassembled into

- factors: $c_{(i_1, i_2)}$, $\Phi_{i_1 q}^{(b,*)}$, $\delta_{i_2 a}$; and
- contraction indices: i_1 , i_2 .

Delta cancellation is carried out in this disassembled form. If there are any factors δ_{jk} or δ_{kj} such that j is a contraction index, that factor is removed along with the contraction index j , and a $j \mapsto k$ index substitution is applied to all the remaining factors. This is repeated as long as applicable, and the repetition trivially terminates at latest when running out of contraction indices or delta factors.

This cancellation step is only applicable once to the example above, and it leaves us with factors $c_{(i_1, a)}$ and $\Phi_{i_1 q}^{(b,*)}$ as well as contraction index i_1 . Therefore, the optimised tensor product becomes

$$j_q = \sum_{i_1} c_{(i_1, a)} \Phi_{i_1 q}^{(b,*)}. \quad (6.4)$$

As one can see, this only needs the tabulation matrix of the scalar element, and each entry of the Jacobian only uses the relevant section of the array c .

6.3 Sum factorisation: Laplace operator

Sum factorisation is a well-known technique first proposed by Orszag [1980] that drastically reduces the algorithmic complexity of finite element assembly for high-order discretisations. To demonstrate this technique, consider the element local tensor of the Laplace operator which is

$$\mathbf{A}_{ij}^K = \sum_{q=1}^{N_q} w_q \left(J_K^{-T}(\xi_q) \hat{\nabla} \Psi_i(\xi_q) \cdot J_K^{-T}(\xi_q) \hat{\nabla} \Psi_j(\xi_q) \right) |\det(J_K(\xi_q))| \quad (6.5)$$

after changing to reference space, equivalent to eq. (2.30) with $\kappa \equiv 1$. For brevity, we introduce the abbreviations $G_q := [J_K(\xi_q)]^{-1}$ and $S_q := |\det(J_K(\xi_q))|$, so we get

$$\mathbf{A}_{ij}^K = \sum_{q=1}^{N_q} w_q S_q \left(G_q^T \hat{\nabla} \Psi_i(\xi_q) \right) \cdot \left(G_q^T \hat{\nabla} \Psi_j(\xi_q) \right). \quad (6.6)$$

Let the reference cell $\hat{K}$ be a square: this simplifies our notation, although TSFC supports higher dimensions uniformly. During UFL preprocessing, matrix-vector multiplications and the inner product in eq. (6.6) are rewritten as

$$\mathbf{A}_{ij}^K = \sum_{q=1}^{N_q} w_q S_q \sum_{k=1}^2 \left(\sum_{l=1}^2 G_{qlk} \frac{\partial \Psi_i}{\partial \hat{x}_l}(\xi_q) \right) \left(\sum_{m=1}^2 G_{qmk} \frac{\partial \Psi_j}{\partial \hat{x}_m}(\xi_q) \right). \quad (6.7)$$

To be able to demonstrate a more generic case, we expand the geometric sums:

$$\begin{aligned} \mathbf{A}_{ij}^K = \sum_{q=1}^{N_q} w_q S_q \left\{ \left(G_{q11} \frac{\partial \Psi_i}{\partial \hat{x}_1}(\xi_q) + G_{q21} \frac{\partial \Psi_i}{\partial \hat{x}_2}(\xi_q) \right) \left(G_{q11} \frac{\partial \Psi_j}{\partial \hat{x}_1}(\xi_q) + G_{q21} \frac{\partial \Psi_j}{\partial \hat{x}_2}(\xi_q) \right) \right. \\ \left. + \left(G_{q12} \frac{\partial \Psi_i}{\partial \hat{x}_1}(\xi_q) + G_{q22} \frac{\partial \Psi_i}{\partial \hat{x}_2}(\xi_q) \right) \left(G_{q12} \frac{\partial \Psi_j}{\partial \hat{x}_1}(\xi_q) + G_{q22} \frac{\partial \Psi_j}{\partial \hat{x}_2}(\xi_q) \right) \right\}. \end{aligned} \quad (6.8)$$

With a tensor product element and quadrature rule as introduced in eqs. (2.36) and (2.38), FInAT provides the substitutions

$$\frac{\partial \Psi_i}{\partial \hat{x}_1}(\xi_q) \mapsto D\Psi_{i_1 q_1}^{(1)} \Psi_{i_2 q_2}^{(2)} \quad \text{and} \quad (6.9)$$

$$\frac{\partial \Psi_i}{\partial \hat{x}_2}(\xi_q) \mapsto \Psi_{i_1 q_1}^{(1)} D\Psi_{i_2 q_2}^{(2)} \quad (6.10)$$

where $i = (i_1, i_2)$, and $\Psi^{(1)}$ and $\Psi^{(2)}$ are numerical tabulation matrices of the one-dimensional elements, while $D\Psi^{(1)}$ and $D\Psi^{(2)}$ are tabulations of the first derivative respectively. Thus eq. (6.8) becomes

$$\begin{aligned} A_{(i_1, i_2)(j_1, j_2)}^K = \sum_{q_1, q_2} w_{q_1}^{(1)} w_{q_2}^{(2)} S_{q_1 q_2} \left\{ \left(G_{q_1 q_2 11} D\Psi_{i_1 q_1}^{(1)} \Psi_{i_2 q_2}^{(2)} + G_{q_1 q_2 21} \Psi_{i_1 q_1}^{(1)} D\Psi_{i_2 q_2}^{(2)} \right) \right. \\ \left(G_{q_1 q_2 11} D\Psi_{j_1 q_1}^{(1)} \Psi_{j_2 q_2}^{(2)} + G_{q_1 q_2 21} \Psi_{j_1 q_1}^{(1)} D\Psi_{j_2 q_2}^{(2)} \right) \\ + \left(G_{q_1 q_2 12} D\Psi_{i_1 q_1}^{(1)} \Psi_{i_2 q_2}^{(2)} + G_{q_1 q_2 22} \Psi_{i_1 q_1}^{(1)} D\Psi_{i_2 q_2}^{(2)} \right) \\ \left. \left(G_{q_1 q_2 12} D\Psi_{j_1 q_1}^{(1)} \Psi_{j_2 q_2}^{(2)} + G_{q_1 q_2 22} \Psi_{j_1 q_1}^{(1)} D\Psi_{j_2 q_2}^{(2)} \right) \right\}. \end{aligned} \quad (6.11)$$

Now we reached the intermediate representation of TSFC: all finite element basis functions have been replaced with tensor algebra expressions. Before we can apply sum factorisation, we need to apply *argument factorisation* to eq. (6.11). Argument factorisation transforms the expression to a sum-of-products form, such that no factor in any product depends on more than one of the free indices of form expression. In the above example, the free indices are i_1 , i_2 , j_1 , and j_2 . Equation (6.11) after argument factorisation looks like

$$\begin{aligned} A_{(i_1, i_2)(j_1, j_2)}^K = \sum_{q_1, q_2} \left\{ D\Psi_{i_1 q_1}^{(1)} \Psi_{i_2 q_2}^{(2)} D\Psi_{j_1 q_1}^{(1)} \Psi_{j_2 q_2}^{(2)} P_{q_1 q_2}^{(1,1)} \right. \\ + D\Psi_{i_1 q_1}^{(1)} \Psi_{i_2 q_2}^{(2)} \Psi_{j_1 q_1}^{(1)} D\Psi_{j_2 q_2}^{(2)} P_{q_1 q_2}^{(1,2)} \\ + \Psi_{i_1 q_1}^{(1)} D\Psi_{i_2 q_2}^{(2)} D\Psi_{j_1 q_1}^{(1)} \Psi_{j_2 q_2}^{(2)} P_{q_1 q_2}^{(2,1)} \\ \left. + \Psi_{i_1 q_1}^{(1)} D\Psi_{i_2 q_2}^{(2)} \Psi_{j_1 q_1}^{(1)} D\Psi_{j_2 q_2}^{(2)} P_{q_1 q_2}^{(2,2)} \right\} \end{aligned} \quad (6.12)$$

where $P_{q_1 q_2}^{(1,1)}$, $P_{q_1 q_2}^{(1,2)}$, $P_{q_1 q_2}^{(2,1)}$, and $P_{q_1 q_2}^{(2,2)}$ aggregate factors that do not depend on any of the free indices. Their precise definitions are:

$$P_{q_1 q_2}^{(1,1)} := w_{q_1}^{(1)} w_{q_2}^{(2)} S_{q_1 q_2} (G_{q_1 q_2 11} G_{q_1 q_2 11} + G_{q_1 q_2 12} G_{q_1 q_2 12}) \quad (6.13a)$$

$$P_{q_1 q_2}^{(1,2)} := w_{q_1}^{(1)} w_{q_2}^{(2)} S_{q_1 q_2} (G_{q_1 q_2 11} G_{q_1 q_2 12} + G_{q_1 q_2 12} G_{q_1 q_2 22}) \quad (6.13b)$$

$$P_{q_1 q_2}^{(2,1)} := w_{q_1}^{(1)} w_{q_2}^{(2)} S_{q_1 q_2} (G_{q_1 q_2 11} G_{q_1 q_2 12} + G_{q_1 q_2 12} G_{q_1 q_2 22}) \quad (6.13c)$$

$$P_{q_1 q_2}^{(2,2)} := w_{q_1}^{(1)} w_{q_2}^{(2)} S_{q_1 q_2} (G_{q_1 q_2 21} G_{q_1 q_2 21} + G_{q_1 q_2 22} G_{q_1 q_2 22}). \quad (6.13d)$$

Were there no restrictions on the factors, any expression would trivially be in a sum-of-products form. One can reach the argument factorised form through the mechanical application of the distributive property $a(b + c) \rightarrow ab + ac$ on products which have

any factor with more than one free index. This rewriting always succeeds since the variational form is by definition linear in its arguments.

To continue with sum factorisation, let us concentrate on just one of the products for now. For example:

$$\sum_{q_1, q_2} D\Psi_{i_1 q_1}^{(1)} \Psi_{i_2 q_2}^{(2)} D\Psi_{j_1 q_1}^{(1)} \Psi_{j_2 q_2}^{(2)} P_{q_1 q_2}^{(1,1)} \quad (6.14)$$

This requires $O(N_{q_1} N_{q_2} N_{f_1} N_{f_2} N_{f_1} N_{f_2})$ floating-point operations to evaluate for all i_1 , i_2 , j_1 , and j_2 . If the finite element is of polynomial order n in both directions, then $N_{f_1} = N_{f_2} = n+1$, and also $N_{q_1} = N_{q_2} = O(n)$. This means $O(n^6)$ operations. Rearranging eq. (6.14) as

$$\sum_{q_1=1}^{N_{q_1}} D\Psi_{i_1 q_1}^{(1)} D\Psi_{j_1 q_1}^{(1)} \left(\sum_{q_2=1}^{N_{q_2}} \Psi_{i_2 q_2}^{(2)} \Psi_{j_2 q_2}^{(2)} P_{q_1 q_2}^{(1,1)} \right) \quad (6.15)$$

only requires $O(N_{q_1} N_{q_2} N_{f_2}^2 + N_{q_1} N_{f_1}^2 N_{f_2}^2)$ operations. The first contribution corresponds to the cost of calculating the expression in brackets for all i_2 , j_2 and q_1 , and the other contribution corresponds to the cost of contracting the resulting tensor over q_1 with the rest of the expression for all i_1 , i_2 , j_1 , and j_2 . Similarly, rearranging eq. (6.14) as

$$\sum_{q_2=1}^{N_{q_2}} \Psi_{i_2 q_2}^{(2)} \Psi_{j_2 q_2}^{(2)} \left(\sum_{q_1=1}^{N_{q_1}} D\Psi_{i_1 q_1}^{(1)} D\Psi_{j_1 q_1}^{(1)} P_{q_1 q_2}^{(1,1)} \right) \quad (6.16)$$

only requires $O(N_{q_1} N_{q_2} N_{f_1}^2 + N_{q_2} N_{f_1}^2 N_{f_2}^2)$ operations. Both arrangements imply $O(n^5)$ operations. Since the number of products in the argument factorised form only depends on the original weak form, but not on the polynomial order n , it is easy to see that assembling a bilinear form on a single quadrilateral requires $O(n^6)$ operations naïvely, or $O(n^5)$ operations with sum factorisation. Generally for a d -cube the operation count is $O(n^{3d})$ naïvely, and $O(n^{2d+1})$ with sum factorisation.

Finally, the only missing piece is a systematic algorithm that takes a tensor product such as eq. (6.14), and rearranges it into an optimal tensor product such as either eq. (6.15) or eq. (6.16). We first disassemble the tensor product into factors and contraction indices as in section 6.2, and then build an optimised tensor product from them. This problem has also been relevant in quantum chemistry applications, and is known as *single-term optimisation* in the literature. Lam et al. [1996] prove that this problem is NP-complete. Their algorithm explores combinations of factors to build a product tree, applying contractions on the way, and pruning the search tree to avoid traversing redundant

Algorithm 10 Building an optimal tensor product

```

1: function MAKE_TENSOR_PRODUCT(factors, indices)
2:   result  $\leftarrow$  nil
3:   minops  $\leftarrow \infty$ 
4:   for all ordering  $\in$  PERMUTATIONS(indices) do
5:     terms  $\leftarrow$  factors
6:     flops  $\leftarrow$  0
7:     for index in ordering do ▷ apply contraction, one index at a time
8:       contract  $\leftarrow \{t \in \text{terms} \mid \text{index} \in \text{free indices of } t\}$ 
9:       deferred  $\leftarrow \{t \in \text{terms} \mid \text{index} \notin \text{free indices of } t\}$ 

10:      product, cost  $\leftarrow$  MAKE_PRODUCT(contract)
11:      term  $\leftarrow$  IndexSum(product, (index,))

12:      terms  $\leftarrow$  deferred  $\cup \{\text{term}\}$ 
13:      flops  $\leftarrow$  flops + cost +  $\prod_{i \in \text{free indices of product}} i.\text{extent}$ 
14:    end for
15:    expr, cost  $\leftarrow$  MAKE_PRODUCT(terms)
16:    flops  $\leftarrow$  flops + cost

17:    if flops < minops then
18:      result  $\leftarrow$  expr
19:      minops  $\leftarrow$  flops
20:    end if
21:  end for
22:  return result
23: end function

```

expressions. Our approach, described in Algorithm 10, orders the contraction indices first: since we never have more than three of them, traversing all permutations is cheap.

The construction of an optimised tensor product expression happens in lines 5 to 16 of Algorithm 10. We apply contractions one index at a time. For example, let us contract along q_1 first. Then the set of factors are split based on dependence on q_1 (lines 8 and 9). The factors in deferred are “pulled out” of $\sum_{q_1}$, while a product expression is constructed from the factors in contract using MAKEPRODUCT, followed by contraction over q_1 . MAKEPRODUCT is a utility function for building a product expression tree from a set of factors, also returning the number of multiplications required to evaluate that product. For further gains, this function may associate factors in a optimised way, but for sum factorisation a trivial product builder suffices. For our example of constructing eq. (6.16),

$$\text{deferred} = \{\Psi_{i_2 q_2}^{(2)}, \Psi_{j_2 q_2}^{(2)}\} \quad \text{and} \quad (6.17)$$

$$\text{term} = \sum_{q_1} D\Psi_{i_1 q_1}^{(1)} D\Psi_{j_1 q_1}^{(1)} P_{q_1 q_2}^{(1,1)}. \quad (6.18)$$

The new tensor contraction expression (term) is added to the deferred factors to produce the set of factors (terms) for the next iteration (line 12). In the next iteration we contract over q_2 and finally get eq. (6.16). For generality, we also handle the case in lines 15 to 16 when multiple factors remain after applying all contraction indices. This construction is repeated for each permutation of the contraction indices, and the expression requiring the lowest number of floating-point operations is selected. Looping over all permutations is not that bad, since $|\text{indices}| \leq 3$ for all relevant problems.

We shall note that the lowest number of floating-point operations does not generally mean the shortest wall clock time. For example, the factorisations eqs. (6.15) and (6.16) both produce $O(n^5)$ algorithms, but the one with more operations might be faster if, for example, the resulting loops are better aligned with the layout of the data arrays. For lower polynomial degrees, when asymptotic effects do not dominate, the situation is much more complicated: it is likely that in some cases sum factorisation is not beneficial at all, especially since it requires argument factorisation which is known to be counterproductive in certain cases. It is beyond the scope of this thesis to consider all these factors together.

6.4 Sum factorisation: parametrised forms

Partial differential equations are frequently parametrised by prescribed spatial functions, which are called *coefficients* in UFL. We have shown that matrix-free assembly of the operator action is analogous to assembling 1-forms with coefficient functions, so as an example we now consider the action of the Laplace operator. We continue from the formulations in eq. (4.1), but now

$$\hat{u}_K(\hat{x}) = \sum_{i=1}^{N_l} \mathbf{u}_i^K \Psi_i(\hat{x}) \quad (6.19)$$

where $\mathbf{u}^K \in \mathbb{R}^{N_f}$ is an array of local basis function coefficients. Using the abbreviations from eq. (6.6), this results in

$$\mathbf{b}_j^K = \sum_{q=1}^{N_q} w_q S_q \left(G_q^T \sum_{i=1}^{N_l} \mathbf{u}_i^K \hat{\nabla} \Psi_i(\xi_q) \right) \cdot \left(G_q^T \hat{\nabla} \Psi_j(\xi_q) \right) \quad (6.20)$$

where $\mathbf{b}^K$ is the assembled local vector.

The common strategy is to first evaluate the coefficient function as $\bar{u}_q \in \mathbb{R}^d$ at every quadrature point q , then simply use that in the form expression:

$$\bar{u}_q = \sum_{i=1}^{N_l} \mathbf{u}_i^K \hat{\nabla} \Psi_i(\xi_q) \quad (6.21)$$

$$\mathbf{b}_j^K = \sum_{q=1}^{N_q} w_q S_q (G_q^T \bar{u}_q) \cdot (G_q^T \hat{\nabla} \Psi_j(\xi_q)) \quad (6.22)$$

The sum factorisation of eq. (6.22) is basically analogous to that which was shown in section 6.3. Equation (6.21) is even simpler: it is similar to the products of the argument factorised form, so it just needs disassembling and calling Algorithm 10.

Table 6.1 summarises the gains of sum factorisation considering various cell types, both for bilinear forms (matrix assembly) as well as for linear forms such as matrix-free operator actions or right-hand side assembly.

6.5 Delta cancellation: nonzero patterns

In section 6.2, we considered delta cancellation for a common, but simple case; now we explore delta cancellation further. Previously, we could assume that the assembled

Table 6.1: Algorithmic complexity of form assembly on a single cell as a function of polynomial degree n .

(a) Naïve implementation			(b) Sum factorised implementation		
Cell type	1-form	2-form	Cell type	1-form	2-form
quadrilateral	$O(n^4)$	$O(n^6)$	quadrilateral	$O(n^3)$	$O(n^5)$
triangular prism	$O(n^6)$	$O(n^9)$	triangular prism	$O(n^5)$	$O(n^7)$
hexahedron	$O(n^6)$	$O(n^9)$	hexahedron	$O(n^4)$	$O(n^7)$

local tensors were dense; now we are going to see that *Kronecker delta* nodes may cause particular nonzero patterns to appear.

For a concrete example, consider the vector mass form

$$\int_{\Omega} u \cdot v \, dx \quad (6.23)$$

where the trial function u and the test function v are chosen from vector- P_n elements. Going through the usual steps, the integral on cell K is evaluated as

$$A_{ij}^K = \sum_{q=1}^{N_q} w_q S_q \Psi_i(\xi_q) \cdot \Psi_j(\xi_q). \quad (6.24)$$

Recall that the usual translation of vector- P_n elements is

$$[\Psi_i(\xi_q)]_k \mapsto \Psi_{i_1 q}^* \delta_{i_2 k} \quad (6.25)$$

where $i = (i_1, i_2)$ and Ψ^* is the tabulation matrix of the *scalar* P_n element. Applying the substitution and rewriting the dot product we have

$$A_{(i_1, i_2)(j_1, j_2)}^K = \sum_{q=1}^{N_q} w_q S_q \sum_{k=1}^d \Psi_{i_1 q}^* \delta_{i_2 k} \Psi_{j_1 q}^* \delta_{j_2 k}. \quad (6.26)$$

If we apply delta cancellation as described in section 6.2 to this product, we obtain

$$A_{(i_1, i_2)(j_1, j_2)}^K = \sum_{q=1}^{N_q} w_q S_q \Psi_{i_1 q}^* \Psi_{j_1 q}^* \delta_{i_2 j_2}, \quad (6.27)$$

or if we build the tensor product with sum factorisation, even

$$A_{(i_1, i_2)(j_1, j_2)}^K = \delta_{i_2 j_2} \sum_{q=1}^{N_q} w_q S_q \Psi_{i_1 q}^* \Psi_{j_1 q}^*. \quad (6.28)$$

It is clear that A^K has a particular nonzero pattern, that is $A_{(i_1, i_2)(j_1, j_2)}^K \neq 0$ only if $i_2 = j_2$. To exploit this property, we need another delta cancellation step that operates *across the assignment*. That is, the tensor product is disassembled and deltas cancelled with contraction indices as in section 6.2. Then, still in the disassembled form, if there are any factors δ_{jk} or δ_{kj} such that j is a free index of the *return variable*, then that factor is removed and a $j \mapsto k$ index substitution is applied to all the remaining factors as well as the return variable. Again, this is repeated as long as applicable. Finally, the tensor product is rebuilt.

In the example above, the return variable is $A_{(i_1, i_2)(j_1, j_2)}^K$, and its free indices are i_1 , i_2 , j_1 , and j_2 . Cancelling the remaining $\delta_{i_2 j_2}$, we end up with

$$A_{(i_1, i_2)(j_1, i_2)}^K = \sum_{q=1}^{N_q} w_q S_q \Psi_{i_1 q}^* \Psi_{j_1 q}^*. \quad (6.29)$$

Note the change in indexing the left-hand side.

In the general case, however, we will not have a nice product structure as in eq. (6.26). Nevertheless, if we apply argument factorisation as in section 6.3, then delta cancellation as described above can be applied to each product. Note that applying delta cancellation across assignments may cause different parts of the form expression to be “assigned” to different *views* of the return variable. A simple approach to correctly handle this case is to assume that the buffer holding A^K is cleared at the beginning, and then make each “assignment” add to that buffer.

6.6 Splitting Concatenate nodes

Finally, we consider Concatenate nodes which come from enriched elements, and are crucial for the $H(\text{div})$ and $H(\text{curl})$ conforming elements of the Q^- family. For the sake of this discussion, we use the RTCF_2 element as an example. As a corollary of eq. (5.14), this element is built as

$$\text{RTCF}_2 = \text{HDiv}(\mathbf{P}_2 \otimes \mathbf{dP}_1) \oplus \text{HDiv}(\mathbf{dP}_1 \otimes \mathbf{P}_2). \quad (6.30)$$

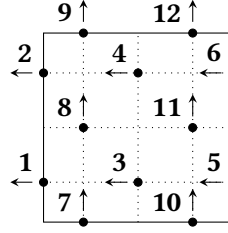


Figure 6.1: Degrees of freedom of the RTCF_2 element.

Its degrees of freedom consist of a 3×2 matrix for $P_2 \otimes dP_1$, and a 2×3 matrix for $dP_1 \otimes P_2$; a total of 12 degrees of freedom. This is graphically shown in Fig. 6.1.

Suppose the quadrature rule is a tensor product rule with the same interval rule in both directions, then FInAT gives the tabulation expression:

$$\text{Concatenate} \left(\left[(-\Psi_{i_1 q_1} \Phi_{i_2 q_2}, 0)_c \right]_{(i_1, i_2)}, \left[(0, \Phi_{i_3 q_1} \Psi_{i_4 q_2})_c \right]_{(i_3, i_4)} \right)_I \quad (6.31)$$

where

- Ψ and Φ are the tabulation matrices of the P_2 and dP_1 interval elements respectively.
- q_1 and q_2 are quadrature indices with the same extent.
- i_1, i_2, i_3 , and i_4 are basis function indices of the subelements. Their extents are 3, 2, 2, and 3 respectively.
- c is the value index. (RTCF₂ element is vector valued.)
- I is the flat basis function index, with extent 12.

Suppose we have a buffer b for the degrees of freedom, and that the above GEM expression is associated with the indexed buffer b_I . The principal approach to “implementing” Concatenate nodes is to split them in combination with the indexed buffer expression. Splitting eq. (6.31) with b_I gives the following pairs:

$$b_{i_2+2(i_1-1)} : (-\Psi_{i_1 q_1} \Phi_{i_2 q_2}, 0)_c \quad (6.32)$$

$$b_{6+i_4+3(i_3-1)} : (0, \Phi_{i_3 q_1} \Psi_{i_4 q_2})_c \quad (6.33)$$

where $b_{i_2+2(i_1-1)}$ indexes the first 6 entries of b as if it were 3×2 matrix indexed by (i_1, i_2) , and $b_{6+i_4+3(i_3-1)}$ indexes the last 6 entries of b as if it were 2×3 matrix indexed

by (i_3, i_4) . These indexed buffer expressions are easily generated: inspecting the shapes of the Concatenate node's operands, we see which segment of b corresponds to each operand and how that should be “reshaped”.

First, we discuss the splitting of Concatenate nodes which originate from the translation of trial and test functions. Suppose we have a bilinear form such as $\int_{\Omega} (\nabla \times u) \cdot (\nabla \times v) dx$. Following the transformation to reference space, the assignment to the local tensor A^K can be written as

$$A_{I,J}^K := \mathcal{F} \left(\Psi_I, \hat{\nabla} \Psi_I, \Psi_J, \hat{\nabla} \Psi_J \right) \quad 1 \leq I \leq N_f \text{ and } 1 \leq J \leq N_f \quad (6.34)$$

where $\{\Psi_I\}_{I=1}^{N_f}$ is the basis of a finite element, I and J are basis function indices, and $\mathcal{F}$ is a functional. Sometimes generality may require second and further derivatives. Suppose the RTCF₂ element was chosen as above, so $N_f = 12$. Similarly, let $\{\Psi_{i_1 i_2}^x\}_{(i_1, i_2)}$ be the basis of $\text{HDiv}(\mathbb{P}_2 \otimes d\mathbb{P}_1)$ and $\{\Psi_{i_3 i_4}^y\}_{(i_3, i_4)}$ be the basis of $\text{HDiv}(d\mathbb{P}_1 \otimes \mathbb{P}_2)$. Note that

$$\{\Psi_{i_1 i_2}^x\}_{(i_1, i_2)} = \{\Psi_I\}_{I=1}^6 \quad \text{and} \quad (6.35)$$

$$\{\Psi_{i_3 i_4}^y\}_{(i_3, i_4)} = \{\Psi_I\}_{I=7}^{12}. \quad (6.36)$$

Equation (6.34) can be trivially split into four assignments, for the combinations of ranges $1 \leq I \leq 6$ and $7 \leq I \leq 12$ with $1 \leq J \leq 6$ and $7 \leq J \leq 12$. Then the substitutions of eqs. (6.35) and (6.36) are directly applicable, which gives:

$$A_{i_2+2(i_1-1), j_2+2(j_1-1)}^K := \mathcal{F} \left(\Psi_{i_1 i_2}^x, \hat{\nabla} \Psi_{i_1 i_2}^x, \Psi_{j_1 j_2}^x, \hat{\nabla} \Psi_{j_1 j_2}^x \right) \quad (6.37a)$$

$$A_{i_2+2(i_1-1), 6+j_4+3(j_3-1)}^K := \mathcal{F} \left(\Psi_{i_1 i_2}^x, \hat{\nabla} \Psi_{i_1 i_2}^x, \Psi_{j_3 j_4}^y, \hat{\nabla} \Psi_{j_3 j_4}^y \right) \quad (6.37b)$$

$$A_{6+i_4+3(i_3-1), j_2+2(j_1-1)}^K := \mathcal{F} \left(\Psi_{i_3 i_4}^y, \hat{\nabla} \Psi_{i_3 i_4}^y, \Psi_{j_1 j_2}^x, \hat{\nabla} \Psi_{j_1 j_2}^x \right) \quad (6.37c)$$

$$A_{6+i_4+3(i_3-1), 6+j_4+3(j_3-1)}^K := \mathcal{F} \left(\Psi_{i_3 i_4}^y, \hat{\nabla} \Psi_{i_3 i_4}^y, \Psi_{j_3 j_4}^y, \hat{\nabla} \Psi_{j_3 j_4}^y \right) \quad (6.37d)$$

To be precise, TSFC does not inspect the construction of finite elements: this splitting of assignments is carried out at the GEM level, that is, after the substitution of basis function evaluations, thus TSFC only “sees” the occurrences of Concatenate nodes. This discussion, however, helps to demonstrate that the substitution of indexed Concatenate nodes such as eq. (6.31) with their split indexed expressions – like eqs. (6.32) and (6.33) – inside the intermediate form expression, along with the corresponding changes to indexing the result buffer, is a valid transformation that eliminates the Concatenate nodes and recovers any product structure within the subelements.

Lastly, we consider the evaluation of parametrising functions – also known as *coefficients* of the multilinear form – at quadrature points. Recalling eq. (6.19), we can apply a similar separation of basis functions and substitution, that is

$$\hat{u}_K(\hat{x}) = \sum_{i=1}^{12} U_i^K \Psi_i(\hat{x}) = \sum_{i=1}^6 U_i^K \Psi_i(\hat{x}) + \sum_{i=7}^{12} U_i^K \Psi_i(\hat{x}) \quad (6.38)$$

$$= \sum_{i_1, i_2} U_{i_2+2(i_1-1)}^K \Psi_{i_1 i_2}^x(\hat{x}) + \sum_{i_3, i_4} U_{6+i_4+3(i_3-1)}^K \Psi_{i_3 i_4}^y(\hat{x}). \quad (6.39)$$

Having recovered the product structure, sum factorisation is now applicable to each summation separately. Again, we split Concatenate nodes along with indexed buffer expressions, but the latter now corresponds to an array of given numbers rather than the result buffer. Although the indexed Concatenate node is not necessarily outermost in the tabulation expression since the enriched element may not be outermost, this is not problem since all compound elements in chapter 5 are linear in their subelements.

6.7 Order of transformations

In previous subsections we have described a number of algorithms transforming the intermediate representation in TSFC. We finally list all transformations in the order of their application:

- (1) Split Concatenate nodes, see section 6.6.
- (2) Apply argument factorisation, see section 6.3.
- (3) Apply delta cancellation with tensor contractions, see section 6.2.
- (4) Apply delta cancellation across assignments, see section 6.5.
- (5) Apply sum factorisation, see section 6.3.

Argument factorisation created a sum-of-products form, sum factorisation is applied on each product, but with the same ordering of contraction indices for all products, to leave more opportunities for factorisation based on the distributivity rule.

- (6) At each contraction level during sum factorisation: apply the ILP factorisation algorithm from COFFEE [Luporini et al., 2017]. This factorisation is based on the

distributivity rule, and especially improves bilinear forms (matrix assembly). It aims to factorise the expression in a way that minimises the operation count in the innermost (hottest) loop. The optimisation problem is solved using an *integer linear programming* (ILP) formulation.

TSFC has several optimisation *modes*, which share the same UFL-to-GEM and GEM-to-C stages, but carry out different GEM-to-GEM transformations in between. Currently, TSFC offers the following modes:

- spectral mode applies all passes listed above. (The current default.)
- coffee mode implements a simplified version of the algorithm developed by [Luporini et al. \[2017\]](#). The simplification is that this mode unconditionally argument factorises, so it contains passes (1), (2), and (6). (The previous default.)
- vanilla mode aims to do as little as possible, so it only applies pass (1) since Concatenate nodes must be removed before the GEM-to-C stage. This mode corresponds to the original behaviour of TSFC described in [Homolya et al. \[2017b\]](#) when most optimisations were done in COFFEE [[Luporini et al., 2015, 2017](#)].
- tensor mode has two unique passes:
 - (7) Flatten Concatenate nodes, destroying their inner structure.
 - (8) Attempt to refactorise the integrand expression such that quadrature is pre-evaluated at compile time. This mimics the older *tensor representation* in FFC [[Kirby and Logg, 2006, 2007](#)], which is highly-performant for low-degree, constant-coefficient bilinear forms on affine simplices.

We must also note that modes have no effect on the evaluation of parametrising functions. For this purpose, we always apply passes (1) and (3), as well as sum factorisation according to section 6.4.

Chapter 7

Code generation: GEM $\rightarrow$ C

This chapter explains the second stage of TSFC, which is the translation of GEM to C, a programming language with explicit loops and only scalar expressions. First, we eliminate the duplicity of *shape* and *free indices* (section 7.1), then present a simple approach to code generation (section 7.2). In section 7.3 we order the generated loop nests such that *loop fusion* can greatly reduce the memory requirement (see section 6.1). Finally, sections 7.4 and 7.5 describe optimisations related to code generation. The content of this chapter is derived from Homolya et al. [2017b].

7.1 Elimination of ComponentTensor nodes

The duplicity of *shape* and *free indices* would noticeably complicate the process of code generation. To avoid that, we eliminate non-scalar shape in non-terminal nodes by removing all ComponentTensor nodes using a rule that is analogous to *beta reduction*:

$$\text{Indexed}(\text{ComponentTensor}(e, \alpha), \beta) \rightarrow e|_{\alpha \rightarrow \beta} \quad (7.1)$$

where e is an expression with scalar shape, α is a multi-index containing distinct free indices of e , and β is a multi-index of the same rank as α . $e|_{\alpha \rightarrow \beta}$ denotes a substitution of indices in e , where α_i is replaced with β_i for all i . The rule replaces indexing into a ComponentTensor by a substitution of the indices.

A ComponentTensor node must be immediately inside an Indexed node or at the root of the expression because Indexed is the only GEM operator that accepts non-scalar shape. Since the UFL $\rightarrow$ GEM stage produces a GEM expression with scalar shape and the argument indices as free indices, the application of eq. (7.1) removes

all ComponentTensors from the intermediate representation. The downside of this approach is the possibility of code duplication when different substitutions are applied on the same subexpression.

To demonstrate this, we continue with our Laplace operator example. To simplify our presentation in section 4.4, most ComponentTensor nodes were eliminated except for K defined in eq. (4.7). Eliminating K in eq. (4.11) gives

$$\sum_q w_q |d| \sum_{i_5} \left(\sum_{i_3} \frac{B_{i_3, i_5}}{d} \begin{bmatrix} E_{q,k}^{(1)} & E_{q,k}^{(2)} \end{bmatrix}_{i_3} \right) \left(\sum_{i_4} \frac{B_{i_4, i_5}}{d} \begin{bmatrix} E_{q,j}^{(1)} & E_{q,j}^{(2)} \end{bmatrix}_{i_4} \right). \quad (7.2)$$

As one can see, this step duplicates the element-wise division of the adjugate matrix by the determinant.

7.2 Generation of loop nests

Since ComponentTensor nodes are eliminated, we can now restrict our attention to the index notation. A simple code generation approach for the evaluation of such tensor expressions is to introduce a *temporary variable* and generate a pair of *perfect loop nests* (Fig. 7.1a) for each tensor contraction (IndexSum). The first loop nest initialises the temporary, and the second accumulates the result. Consequently, the first loop nest contains all the free indices of the IndexSum expression, while the second loop nest has one more index, the index being summed over. For example, applying this approach to eq. (7.2), we introduce temporaries $J^{(1,1)}$, $J^{(1,2)}$, $J^{(2,1)}$, $J^{(2,2)}$, t_1 , t_2 , t_3 and A . The evaluation code is presented in Algorithm 11, which also adds loop nests for “defining” ListTensor expressions, such as adjugate matrix B , $l_1 = \begin{bmatrix} E_{q,k}^{(1)} & E_{q,k}^{(2)} \end{bmatrix}$, and $l_2 = \begin{bmatrix} E_{q,j}^{(1)} & E_{q,j}^{(2)} \end{bmatrix}$.

While this simple approach produces correct code, introducing temporaries merely for tensor contractions may cause too much *inlining*. In particular, shared subexpressions other than tensor contractions are still expanded, and there is also room for *loop hoisting* (Fig. 7.1). Therefore, we also introduce temporaries to evaluate subexpressions that a) have a *reference count* greater than one, or b) have a different set of *free indices* than their unique parent. In our example, the determinant of the Jacobian is an instance of a), and $w_q * \text{abs}(\dots)$ in the final loop nest is an instance of b). So we introduce temporaries d and t_4 respectively. The result of this is Algorithm 12.

Finally, we apply *loop fusion* to reduce the size of the temporaries (see Fig. 7.2). Here we use a simple and fast algorithm. First, we construct an ordering of all free indices

Algorithm 11 Simple code generation for the expression eq. (7.2).

<pre> 1: for all q do 2: $J^{(1,1)}[q] \leftarrow 0$ 3: for all q, r do 4: $J^{(1,1)}[q] += C_{q,r}^{(1,1)} * c_r$ 5: for all q do 6: $J^{(1,2)}[q] \leftarrow 0$ 7: for all q, r do 8: $J^{(1,2)}[q] += C_{q,r}^{(1,2)} * c_r$ 9: for all q do 10: $J^{(2,1)}[q] \leftarrow 0$ 11: for all q, r do 12: $J^{(2,1)}[q] += C_{q,r}^{(2,1)} * c_r$ 13: for all q do 14: $J^{(2,2)}[q] \leftarrow 0$ 15: for all q, r do 16: $J^{(2,2)}[q] += C_{q,r}^{(2,2)} * c_r$ 17: for all q do 18: $B[q, \dots] \leftarrow \begin{bmatrix} J^{(2,2)}[q] & -J^{(1,2)}[q] \\ -J^{(2,1)}[q] & J^{(1,1)}[q] \end{bmatrix}$ 19: for all q, k do 20: $l_1[q, k, \dots] \leftarrow \begin{bmatrix} E_{q,k}^{(1)} & E_{q,k}^{(2)} \end{bmatrix}$ </pre>	<pre> 21: for all i_5, q, k do 22: $t_1[i_5, q, k] \leftarrow 0$ 23: for all i_3, i_5, q, k do 24: $t_1[i_5, q, k] += B[q, i_3, i_5] \div (J^{(1,1)}[q] * J^{(2,2)}[q] - J^{(1,2)}[q] * J^{(2,1)}[q]) * l_1[q, k, i_3]$ 25: for all q, j do 26: $l_2[q, j, \dots] \leftarrow \begin{bmatrix} E_{q,j}^{(1)} & E_{q,j}^{(2)} \end{bmatrix}$ 27: for all i_5, q, j do 28: $t_2[i_5, q, j] \leftarrow 0$ 29: for all i_4, i_5, q, j do 30: $t_2[i_5, q, j] += B[q, i_4, i_5] \div (J^{(1,1)}[q] * J^{(2,2)}[q] - J^{(1,2)}[q] * J^{(2,1)}[q]) * l_2[q, j, i_4]$ 31: for all q, j, k do 32: $t_3[q, j, k] \leftarrow 0$ 33: for all i_5, q, j, k do 34: $t_3[q, j, k] += t_1[i_5, q, k] * t_2[i_5, q, j]$ 35: for all j, k do 36: $A[j, k] \leftarrow 0$ 37: for all j, k, q do 38: $A[j, k] += w_q * \text{abs}(J^{(1,1)}[q] * J^{(2,2)}[q] - J^{(1,2)}[q] * J^{(2,1)}[q]) * t_3[q, j, k]$ </pre>
--	--

```

1: for all  $i$  do
2:   for all  $j$  do
3:      $d[i, j] \leftarrow (a[i] + b[i]) * c[j]$ 

```

(a) A *perfect loop nest* of two loops: the inner loop is perfectly nested in the outer loop since the outer loop contains no other statements than the inner loop.

```

1: for all  $i$  do
2:    $t[i] \leftarrow a[i] + b[i]$ 
3:   for all  $j$  do
4:      $d[i, j] \leftarrow t[i] * c[j]$ 

```

(b) The result of hoisting operations from the inner loop; an *imperfect* loop nest.

Figure 7.1: An example of *loop hoisting*. The calculation $a[i] + b[i]$ is independent of the inner loop, thus it need not be repeated for all j .

Algorithm 12 Introducing further temporaries in Algorithm 11.

```

... skipped ...
23: for all  $q$  do                                     ▶ Common subexpression
24:    $d[q] = J^{(1,1)}[q] * J^{(2,2)}[q] - J^{(1,2)}[q] * J^{(2,1)}[q]$ 
25:   for all  $i_3, i_5, q, k$  do
26:      $t_1[i_5, q, k] += B[q, i_3, i_5] \div d[q] * l_1[q, k, i_3]$ 
27:   for all  $q, j$  do
28:      $l_2[q, j, \dots] \leftarrow \begin{bmatrix} E_{q,j}^{(1)} & E_{q,j}^{(2)} \end{bmatrix}$ 
29:   for all  $i_5, q, j$  do
30:      $t_2[i_5, q, j] \leftarrow 0$ 
31:   for all  $i_4, i_5, q, j$  do
32:      $t_2[i_5, q, j] += B[q, i_4, i_5] \div d[q] * l_2[q, j, i_4]$ 
33:   for all  $q, j, k$  do
34:      $t_3[q, j, k] \leftarrow 0$ 
35:   for all  $i_5, q, j, k$  do
36:      $t_3[q, j, k] += t_1[i_5, q, k] * t_2[i_5, q, j]$ 
37:   for all  $j, k$  do
38:      $A[j, k] \leftarrow 0$ 
39:   for all  $q$  do                                     ▶ Hoisted computation
40:      $t_4[q] = w_q * \text{abs}(d[q])$ 
41:   for all  $j, k, q$  do
42:      $A[j, k] += t_4[q] * t_3[q, j, k]$ 

```

<pre> 1: for all i do 2: $c[i] \leftarrow a[i] + b[i]$ 3: for all i do 4: $d[i] \leftarrow c[i] * c[i]$ </pre>	<pre> 1: for all i do 2: $c[i] \leftarrow a[i] + b[i]$ 3: $d[i] \leftarrow c[i] * c[i]$ </pre>	<pre> 1: for all i do 2: $c \leftarrow a[i] + b[i]$ 3: $d[i] \leftarrow c * c$ </pre>
(a) Sequence of two loops	(b) After loop fusion	(c) After shape reduction

Figure 7.2: An example of a pair of loops before and after *loop fusion*. If c is not needed later, its shape can be reduced after fusion.

<pre> 1: for all i do 2: for all j do 3: $c[i, j] \leftarrow a[i] * b[j]$ </pre>	<pre> 1: for all j do 2: for all i do 3: $c[i, j] \leftarrow a[i] * b[j]$ </pre>
---	---

Figure 7.3: An example of *loop interchange*: the inner and outer loops are swapped. Loop interchange is only applicable to a perfect loop nest (see Fig. 7.1a).

that appear in the GEM expression, and apply this ordering to each (perfect) loop nest. Based on our knowledge about the structure of finite element assembly kernels, and in common with FFC, we choose the *quadrature index* to be the first (outermost), followed by the *argument indices* in order, and then by all other indices. A possible ordering that we use for our example is $(q, j, k, r, i_3, i_4, i_5)$. This means a number of *loop interchanges* (Fig. 7.3) in Algorithm 12. Considering dependencies, we also take a *topological ordering* of the loop nests that heuristically tries to maximise loop fusion (section 7.3). For our example, this results in Algorithm 13. Finally, we apply maximal loop fusion without further reordering. The fused algorithm is Algorithm 14. Note that many of the temporaries are now scalar, or at least have a reduced tensor order.

Internally, these loop transformations are implemented to work on a representation in *Impero*, an abstract syntax tree format that helps the generation of imperative code. Impero is not a self-contained language; it relies on GEM, and Impero nodes may contain references to GEM nodes corresponding to subexpressions of the GEM representation. However, it makes the loop structure explicit, which is precisely what we need here.

Listing 1 is a straightforward transcription of Algorithm 14 to C code. The definitions of compile-time constant tensors $C^{(1,1)}$, $C^{(1,2)}$, $C^{(2,1)}$, $C^{(2,2)}$, w , $E^{(1)}$, and $E^{(2)}$ is prepended to the function body. The element tensor A and the local coordinate coefficients vector c are defined as kernel arguments. Lines 10 to 14 may be omitted if the caller clears A .

Algorithm 13 Reordering loops in Algorithm 12.

```

1: for all  $j, k$  do
2:    $A[j, k] \leftarrow 0$ 
3: for all  $q$  do
4:    $J^{(1,1)}[q] \leftarrow 0$ 
5: for all  $q$  do
6:    $J^{(1,2)}[q] \leftarrow 0$ 
7: for all  $q$  do
8:    $J^{(2,1)}[q] \leftarrow 0$ 
9: for all  $q$  do
10:   $J^{(2,2)}[q] \leftarrow 0$ 
11: for all  $q, r$  do
12:   $J^{(1,1)}[q] += C_{q,r}^{(1,1)} * c_r$ 
13: for all  $q, r$  do
14:   $J^{(1,2)}[q] += C_{q,r}^{(1,2)} * c_r$ 
15: for all  $q, r$  do
16:   $J^{(2,1)}[q] += C_{q,r}^{(2,1)} * c_r$ 
17: for all  $q, r$  do
18:   $J^{(2,2)}[q] += C_{q,r}^{(2,2)} * c_r$ 
19: for all  $q$  do
20:   $B[q, \dots] \leftarrow \begin{bmatrix} J^{(2,2)}[q] & -J^{(1,2)}[q] \\ -J^{(2,1)}[q] & J^{(1,1)}[q] \end{bmatrix}$ 
21: for all  $q$  do
22:   $d[q] = J^{(1,1)}[q] * J^{(2,2)}[q] - J^{(1,2)}[q] * J^{(2,1)}[q]$ 
23: for all  $q, k$  do
24:   $l_1[q, k, \dots] \leftarrow \begin{bmatrix} E_{q,k}^{(1)} & E_{q,k}^{(2)} \end{bmatrix}$ 
25: for all  $q, k, i_5$  do
26:   $t_1[q, k, i_5] \leftarrow 0$ 
27: for all  $q, k, i_3, i_5$  do
28:   $t_1[q, k, i_5] += B[q, i_3, i_5] \div d[q] * l_1[q, k, i_3]$ 
29: for all  $q$  do
30:   $t_4[q] = w_q * \text{abs}(d[q])$ 
31: for all  $q, j$  do
32:   $l_2[q, j, \dots] \leftarrow \begin{bmatrix} E_{q,j}^{(1)} & E_{q,j}^{(2)} \end{bmatrix}$ 
33: for all  $q, j, i_5$  do
34:   $t_2[q, j, i_5] \leftarrow 0$ 
35: for all  $q, j, i_4, i_5$  do
36:   $t_2[q, j, i_5] += B[q, i_4, i_5] \div d[q] * l_2[q, j, i_4]$ 
37: for all  $q, j, k$  do
38:   $t_3[q, j, k] \leftarrow 0$ 
39: for all  $q, j, k, i_5$  do
40:   $t_3[q, j, k] += t_1[q, k, i_5] * t_2[q, j, i_5]$ 
41: for all  $q, j, k$  do
42:   $A[j, k] += t_4[q] * t_3[q, j, k]$ 

```

Algorithm 14 The result of loop fusion on Algorithm 13.

```

1: for all  $j, k$  do
2:    $A[j, k] \leftarrow 0$ 
3: for all  $q$  do
4:    $J^{(1,1)} \leftarrow 0$ 
5:    $J^{(1,2)} \leftarrow 0$ 
6:    $J^{(2,1)} \leftarrow 0$ 
7:    $J^{(2,2)} \leftarrow 0$ 
8:   for all  $r$  do
9:      $J^{(1,1)} += C_{q,r}^{(1,1)} * c_r$ 
10:     $J^{(1,2)} += C_{q,r}^{(1,2)} * c_r$ 
11:     $J^{(2,1)} += C_{q,r}^{(2,1)} * c_r$ 
12:     $J^{(2,2)} += C_{q,r}^{(2,2)} * c_r$ 
13:     $B = \begin{bmatrix} J^{(2,2)} & -J^{(1,2)} \\ -J^{(2,1)} & J^{(1,1)} \end{bmatrix}$ 
14:     $d = J^{(1,1)} * J^{(2,2)} - J^{(1,2)} * J^{(2,1)}$ 
15:    for all  $k$  do
16:       $l_1 \leftarrow \begin{bmatrix} E_{q,k}^{(1)} & E_{q,k}^{(2)} \end{bmatrix}$ 
17:      for all  $i_5$  do
18:         $t_1[k, i_5] \leftarrow 0$ 
19:        for all  $i_3, i_5$  do
20:           $t_1[k, i_5] += B[i_3, i_5] \div d * l_1[i_3]$ 
21:       $t_4 = w_q * \text{abs}(d)$ 
22:      for all  $j$  do
23:         $l_2 \leftarrow \begin{bmatrix} E_{q,j}^{(1)} & E_{q,j}^{(2)} \end{bmatrix}$ 
24:        for all  $i_5$  do
25:           $t_2[i_5] \leftarrow 0$ 
26:          for all  $i_4, i_5$  do
27:             $t_2[i_5] += B[i_4, i_5] \div d * l_2[i_4]$ 
28:          for all  $k$  do
29:             $t_3 \leftarrow 0$ 
30:            for all  $i_5$  do
31:               $t_3 += t_1[k, i_5] * t_2[i_5]$ 
32:             $A[j, k] += t_4 * t_3$ 

```

Listing 1 Local assembly kernel for $\int \nabla u \cdot \nabla v \, dx$ in C.

```

1  static inline void kernel(double A[4][4], const double c[8])
2  {
3      static const double C11[4][8] = { ... };
4      static const double C12[4][8] = { ... };
5      static const double C21[4][8] = { ... };
6      static const double C22[4][8] = { ... };
7      static const double w[4] = { ... };
8      static const double E1[4][4] = { ... };
9      static const double E2[4][4] = { ... };
10     for (int j = 0; j < 4; j++) {
11         for (int k = 0; k < 4; k++) {
12             A[j][k] = 0;
13         }
14     }
15     for (int q = 0; q < 4; q++) {
16         double J11 = 0;
17         double J12 = 0;
18         double J21 = 0;
19         double J22 = 0;
20         for (int r = 0; r < 8; r++) {
21             J11 += C11[q][r]*c[r];
22             J12 += C12[q][r]*c[r];
23             J21 += C21[q][r]*c[r];
24             J22 += C22[q][r]*c[r];
25         }
26         double B[2][2] = { { J22, -J12 }, { -J21, J11 } };
27         double d = J11*J22 - J12*J21;
28         double t1[4][2];
29         for (int k = 0; k < 4; k++) {
30             double l1[2] = { E1[q][k], E2[q][k] };
31             for (int i_5 = 0; i_5 < 2; i_5++) {
32                 t1[k][i_5] = 0;
33             }
34             for (int i_3 = 0; i_3 < 2; i_3++) {
35                 for (int i_5 = 0; i_5 < 2; i_5++) {
36                     t1[k][i_5] += (B[i_3][i_5]/d)*l1[i_3];
37                 }
38             }
39         }
40         double t4 = w[q]*fabs(d);
41         for (int j = 0; j < 4; j++) {
42             double l2[2] = { E1[q][j], E2[q][j] };
43             double t2[2];
44             for (int i_5 = 0; i_5 < 2; i_5++) {
45                 t2[i_5] = 0;
46             }
47             for (int i_4 = 0; i_4 < 2; i_4++) {
48                 for (int i_5 = 0; i_5 < 2; i_5++) {
49                     t2[i_5] += (B[i_4][i_5]/d)*l2[i_4];
50                 }
51             }
52             for (int k = 0; k < 4; k++) {
53                 double t3 = 0;
54                 for (int i_5 = 0; i_5 < 2; i_5++) {
55                     t3 += t1[k][i_5]*t2[i_5];
56                 }
57                 A[j][k] += t4*t3;
58             }
59         }
60     }

```

7.3 Topological ordering of loop nests

The GEM DAG describes the dependencies of each node in that DAG. The nodes can be evaluated in any order which respects those dependencies. That is, if there is an edge from node a to node b then node b must be evaluated before node a . Any ordering which respects the constraints imposed by the DAG edges is called a *topological ordering*, and the process of generating such an ordering is called *topological sorting*.

Topological sorting is a well-understood problem [Cormen et al., 2001, §22.4]. Every DAG admits a topological ordering, but this is not in general unique. For correct code generation as described in section 7.2, any topological ordering suffices. However, the choice of ordering determines which loops can be fused, and thus how much memory is saved. Reducing the *working set size* can help to fit into faster memory, thus also reducing execution time. To ensure reasonable compilation times, we have implemented a fast algorithm that may produce suboptimal results, rather than striving for optimality at the cost of high algorithmic complexity [Lam et al., 1999b; Allam et al., 2006].

We use a variant of Kahn’s algorithm [Kahn, 1962; Knuth, 1968, §2.2.3, Algorithm T] with heuristics to maximize loop fusion. Algorithm 15 presents Kahn’s algorithm for topological sorting. Line 6 is a point of ambiguity, since S can have more than one element. Any choice will yield a correct, but different, topological ordering, so our heuristics are about choosing “wisely”.

Loop fusion occurs between two consecutive loop nests for all common outer loops. We maximize this at every choice in a greedy manner. The number of loops in the previously chosen loop nest, however, poses an upper limit on the number of loops that can be fused as a result of the current choice. For example, if the previous choice was a loop nest with indices $(q,)$, this rule so far expresses no preference between $(q,)$ and (q, r) for the next choice, because only the q loop can be fused in both cases. Therefore we further refine our heuristic to prefer those loop nests with exactly the same loops as the previously chosen loop nest. Delaying “entry” into further inner loops, is a heuristic designed in the hope that more loop nests with those indices will satisfy the dependency requirements once we have no other option but to *do* “enter” those further inner loops, thus enabling more loop fusion later.

Algorithm 16 shows our version of the CHOOSE-ELEMENT function. We maintain the ordered loop indices of the previously chosen loop nest in *cursor*, which is initialized to $()$ at the beginning of a topological sort. We can assume that S is non-empty. First, we try to choose an element with the same loop indices as the previous one (lines 3 to 5). If that is not possible, we try to choose an element such that *cursor* is a prefix of its loop

Algorithm 15 Kahn's algorithm for *topological sorting*

```

1: function TOPOLOGICAL-SORT( $G$ )
2:    $refcount \leftarrow$  in-degree for each node in  $G$ 
3:    $L \leftarrow []$ 
4:    $S \leftarrow \{u \mid refcount[u] = 0\}$  ▷ root nodes of  $G$ 
5:   while  $S$  is non-empty do
6:      $u \leftarrow$  CHOOSE-ELEMENT( $S$ )
7:      $S \leftarrow S - \{u\}$ 
8:     insert  $u$  to the front of  $L$ 
9:     for all  $v \in children[u]$  do
10:      DECREMENT( $refcount[v]$ )
11:      if  $refcount[v] = 0$  then
12:         $S \leftarrow S \cup \{v\}$ 
13:      end if
14:    end for
15:  end while
16:  return  $L$ 
17: end function

```

Algorithm 16 Heuristic to maximize loop fusion

```

1: function CHOOSE-ELEMENT( $S$ )
2:   while true do
3:      $S_1 = \{u \in S \mid loop\_indices[u] = cursor\}$ 
4:     if  $S_1$  is non-empty then
5:       return any element from  $S_1$ 
6:     end if
7:      $S_2 = \{u \in S \mid cursor \text{ is a prefix of } loop\_indices[u]\}$ 
8:     if  $S_2$  is non-empty then
9:        $u \leftarrow$  any element from  $S_2$ 
10:       $cursor \leftarrow loop\_indices[u]$ 
11:      return  $u$ 
12:     end if
13:     remove last index from  $cursor$ 
14:   end while
15: end function

```

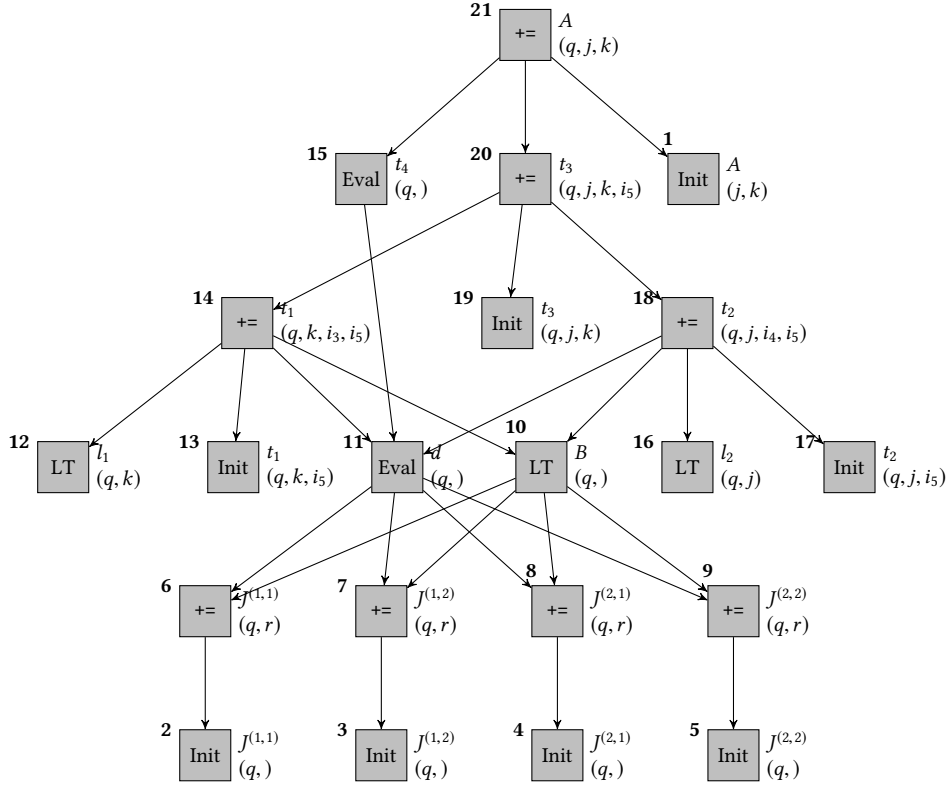


Figure 7.4: *Dependency graph* of the loop nests of Algorithm 12. At the right of each node are the name of the *temporary variable* and the ordered loop indices of the loop nest. The bold numbers at the upper left of each node represent a valid ordering of these operations.

indices (lines 7 to 11), and we update *cursor* accordingly. If that is not possible either, we drop the last (innermost) index from *cursor* and repeat. Before this step *cursor* must have had at least one index, otherwise $S_2 = S \neq \emptyset$, so the function would have returned in line 11.

Fig. 7.4 shows the *dependency graph* of the loop nests in Algorithm 12, along with the solution our algorithm yields. The ordering precisely corresponds to what we have seen in Algorithm 13.

7.4 Unrolling short IndexSum loops

In most cases, unrolling short loops, which are typically geometric loops, improves performance. When this optimisation is applied at the beginning of code generation, the generated code for the Laplace operator becomes as in Algorithm 17. It also negates the code duplication around Jacobian inverses that was caused by the *beta expansion* step (section 7.1). This optimisation is enabled by default in TSFC. However, as demonstrated, it is completely optional, and can be delayed to run after those optimisations that benefit from the tensor structure of the source expression.

7.5 COFFEE integration

Although we did not emphasize this, TSFC does not generate C code directly, but an *abstract syntax tree* (AST) for C, so that COFFEE can further optimize the kernel. Since this is just a different representation of C code, the discussions earlier in this chapter are still completely valid. This was originally to retain the optimisations of Luporini et al. [2015, 2017]. However, since we are moving to apply optimisations at higher levels of abstractions, COFFEE is currently being phased out.

Algorithm 17 Generated code when unrolling applied, equivalent to Algorithm 14.

```

1: for all  $j, k$  do
2:    $A[j, k] \leftarrow 0$ 
3:   for all  $q$  do
4:      $J^{(1,1)} \leftarrow 0$ 
5:      $J^{(1,2)} \leftarrow 0$ 
6:      $J^{(2,1)} \leftarrow 0$ 
7:      $J^{(2,2)} \leftarrow 0$ 
8:     for all  $r$  do
9:        $J^{(1,1)} += C_{q,r}^{(1,1)} * c_r$ 
10:       $J^{(1,2)} += C_{q,r}^{(1,2)} * c_r$ 
11:       $J^{(2,1)} += C_{q,r}^{(2,1)} * c_r$ 
12:       $J^{(2,2)} += C_{q,r}^{(2,2)} * c_r$ 
13:      $d = J^{(1,1)} * J^{(2,2)} - J^{(1,2)} * J^{(2,1)}$ 
14:      $K^{(1,1)} = J^{(2,2)} \div d$ 
15:      $K^{(1,2)} = -J^{(1,2)} \div d$ 
16:      $K^{(2,1)} = -J^{(2,1)} \div d$ 
17:      $K^{(2,2)} = J^{(1,1)} \div d$ 
18:     for all  $k$  do
19:        $t_1[k] = K_{11}E_{q,k}^{(1)} + K_{21}E_{q,k}^{(2)}$ 
20:        $t_2[k] = K_{12}E_{q,k}^{(1)} + K_{22}E_{q,k}^{(2)}$ 
21:      $t_5 = w_q * \text{abs}(d)$ 
22:     for all  $j$  do
23:        $t_3 = K_{11}E_{q,j}^{(1)} + K_{21}E_{q,j}^{(2)}$ 
24:        $t_4 = K_{12}E_{q,j}^{(1)} + K_{22}E_{q,j}^{(2)}$ 
25:       for all  $k$  do
26:          $A[j, k] += t_5 * (t_1[k] * t_3 + t_2[k] * t_4)$ 

```

Chapter 8

Experimental evaluation

We experimentally evaluate form compilation speed as well as the performance of the generated code. Form compilation speed is examined in section 8.2, where our main focus is quick code generation even for highly complicated forms, enabling an interactive development of advanced finite element schemes. Section 8.3 verifies the performance benefits of the work in chapters 5 and 6, and confirms optimal algorithmic complexity for high-order finite element kernels. Full description of the test problems used in these experiments are provided in section 8.1. The content of this chapter is derived from Homolya et al. [2017b,a].

8.1 Test cases

First, we list and briefly explain the bilinear forms used in the experiments.

Laplace operator

$$a(v, u) = \int_{\Omega} \nabla u \cdot \nabla v \, dx \quad (8.1)$$

Helmholtz equation

$$a(v, u) = \int_{\Omega} (uv + \nabla u \cdot \nabla v) \, dx \quad (8.2)$$

Stokes momentum term

$$a(\mathbf{v}, \mathbf{u}) = \int_{\Omega} \nabla \mathbf{u} : \nabla \mathbf{v} \, dx \quad (8.3)$$

where $\mathbf{u}$ and $\mathbf{v}$ are vector-valued trial and test functions respectively.

Curl-curl operator

$$a(\mathbf{v}, \mathbf{u}) = \int_{\Omega} (\nabla \times \mathbf{u}) \cdot (\nabla \times \mathbf{v}) \, dx. \quad (8.4)$$

where $\mathbf{u}$ and $\mathbf{v}$ are vector-valued trial and test functions respectively, taken from $H(\text{curl})$ conforming function spaces.

Linear elasticity

In elastic models the solution is typically a *displacement vector* field. Let $\mathbf{u}$ and $\mathbf{v}$ be vector-valued trial and test functions, and let ε denote the symmetric part of the Jacobian of a vector field. Then the bilinear form is defined as follows:

$$\varepsilon(\mathbf{u}) = \frac{1}{2} [\nabla \mathbf{u} + (\nabla \mathbf{u})^T] \quad (8.5)$$

$$a(\mathbf{v}, \mathbf{u}) = \int_{\Omega} \varepsilon(\mathbf{u}) : \varepsilon(\mathbf{v}) \, dx \quad (8.6)$$

Hyperelasticity

The simplest hyperelastic model is the Saint Venant–Kirchhoff model [Logg et al., 2012a, p. 529]. First, we define the strain energy function over the displacement vector field $\mathbf{u}$:

$$\mathbf{F} = \mathbf{I} + \nabla \mathbf{u} \quad \triangleright \text{Deformation gradient} \quad (8.7)$$

$$\mathbf{C} = \mathbf{F}^T \mathbf{F} \quad \triangleright \text{Right Cauchy-Green tensor} \quad (8.8)$$

$$\mathbf{E} = (\mathbf{C} - \mathbf{I})/2 \quad \triangleright \text{Euler-Lagrange strain tensor} \quad (8.9)$$

$$\Psi = \frac{\lambda}{2} [\text{tr}(\mathbf{E})]^2 + \mu \text{tr}(\mathbf{E}^2) \quad \triangleright \text{Strain energy function} \quad (8.10)$$

where λ and μ are the *Lamé parameters*, and $\mathbf{I}$ is the identity matrix. Now, we define the Piola-Kirchhoff stress tensors:

$$\mathbf{S} = \frac{\partial \Psi}{\partial \mathbf{E}} \quad \triangleright \text{Second Piola-Kirchhoff stress tensor} \quad (8.11)$$

$$\mathbf{P} = \mathbf{F} \mathbf{S} \quad \triangleright \text{First Piola-Kirchhoff stress tensor} \quad (8.12)$$

Through symbolic differentiation UFL derives automatically that $\mathbf{S} = \lambda \text{tr}(\mathbf{E})\mathbf{I} + 2\mu\mathbf{E}$. Finally, the residual form of this nonlinear problem is

$$r(\mathbf{u}; \mathbf{v}) = \mathbf{P}(\mathbf{u}) : \nabla \mathbf{v} - \mathbf{b} \cdot \mathbf{v} \quad (8.13)$$

where $\mathbf{b}$ is the external forcing. To assemble a left-hand side, one must linearise the residual around an *approximate* solution $\mathbf{u}$:

$$a_{\mathbf{u}}(\mathbf{v}, \delta \mathbf{u}) = \int_{\Omega} \delta r(\mathbf{u}; \mathbf{v}, \delta \mathbf{u}) \, dx = \int_{\Omega} \lim_{\epsilon \rightarrow 0} \frac{r(\mathbf{u} + \epsilon \delta \mathbf{u}; \mathbf{v}) - r(\mathbf{u}; \mathbf{v})}{\epsilon} \, dx \quad (8.14)$$

This bilinear form has trial function $\delta \mathbf{u}$, test function $\mathbf{v}$, and $\mathbf{u}$ is a coefficient of the form.

Holzapfel–Ogden model

A more complicated hyperelastic model described in [Balaban et al. \[2016\]](#). Unlike the Saint Venant–Kirchhoff model, this model uses a volume-preserving right Cauchy–Green strain tensor:

$$J = \det(\mathbf{F}) \quad (8.15)$$

$$\bar{\mathbf{C}} = J^{-\frac{2}{3}} \mathbf{F}^T \mathbf{F} \quad (8.16)$$

Then defines the following rotation invariant functions:

$$I_1 = \text{tr}(\bar{\mathbf{C}}) \quad (8.17)$$

$$I_{4f} = \mathbf{e}_f \cdot \bar{\mathbf{C}} \mathbf{e}_f \quad (8.18)$$

$$I_{4s} = \mathbf{e}_s \cdot \bar{\mathbf{C}} \mathbf{e}_s \quad (8.19)$$

$$I_{8fs} = \mathbf{e}_s \cdot \bar{\mathbf{C}} \mathbf{e}_f \quad (8.20)$$

where $\mathbf{e}_f$ and $\mathbf{e}_s$ denote unit vectors pointing in the local myocardial fibre and sheet directions, respectively. The strain energy function is defined as

$$\begin{aligned} \Psi = & \frac{a}{2b} (\exp [b(I_1 - 3)] - 1) \\ & + \frac{h(I_{4f})a_f}{2b_f} (\exp [b_f(I_{4f} - 1)^2] - 1) \\ & + \frac{h(I_{4s})a_s}{2b_s} (\exp [b_s(I_{4s} - 1)^2] - 1) \\ & + \frac{a_{fs}}{2b_{fs}} (\exp [bI_{8fs}^2] - 1) \end{aligned} \quad (8.21)$$

where $h(x)$ is a Heaviside function with a jump at $x = 1$, and the material parameters are

$$m = (a, b, a_f, b_f, a_s, b_s, a_{fs}, b_{fs}).$$

We define the first Piola-Kirchhoff stress tensor, and the residual form without external forcing:

$$\mathbf{P} = \frac{\partial \Psi}{\partial \mathbf{F}} \quad \triangleright \text{Find first Piola-Kirchhoff tensor} \quad (8.22)$$

$$r = \mathbf{P} : \nabla \mathbf{v} \quad (8.23)$$

Finally, eq. (8.14) gives the bilinear form again.

8.2 Form compilation speed

Since one of the core issues with the modified version of FFC in Firedrake was poor form compilation speed for complicated forms and with non-affine mesh geometry (section 2.2), we first experimentally evaluate and compare form compilation times. We measure the following form compilers:

- FFC quadrature representation (supports affine cells only)
- Modified version of FFC quadrature representation, developed for Firedrake to support non-affine cells.
- UFLACS
- TSFC (vanilla mode)

We disabled optional optimisations, and also included in the form compilation times the time to compile the generated C or C++ code, using GCC 5.4.0 as the C and C++ compiler with optimisation level -O2. FFC tensor representation is not included since its limitations are well-known [Ølgaard and Wells, 2010] (section 2.2), and it fails to compile our more difficult test problems due to missing features. Our test problems consist of the left-hand sides of the following four elliptic PDEs of increasing complexity: Helmholtz equation, linear elasticity, hyperelasticity, and the Holzapfel–Ogden model.

Fig. 8.1 presents violin plots of the form compilation times, each made from 18 samples. The measurements were run on a machine with an Intel® Xeon® E5–2620 (Sandy Bridge) 2.00 GHz CPU and 64GB RAM. Due to the nature of development with Firedrake or FEniCS, the form compiler is frequently invoked in an interactive manner. Consequently, form compilation shall preferably take no more time than the fraction of a second; waiting for more than a couple of seconds is undesirable in a natural workflow.

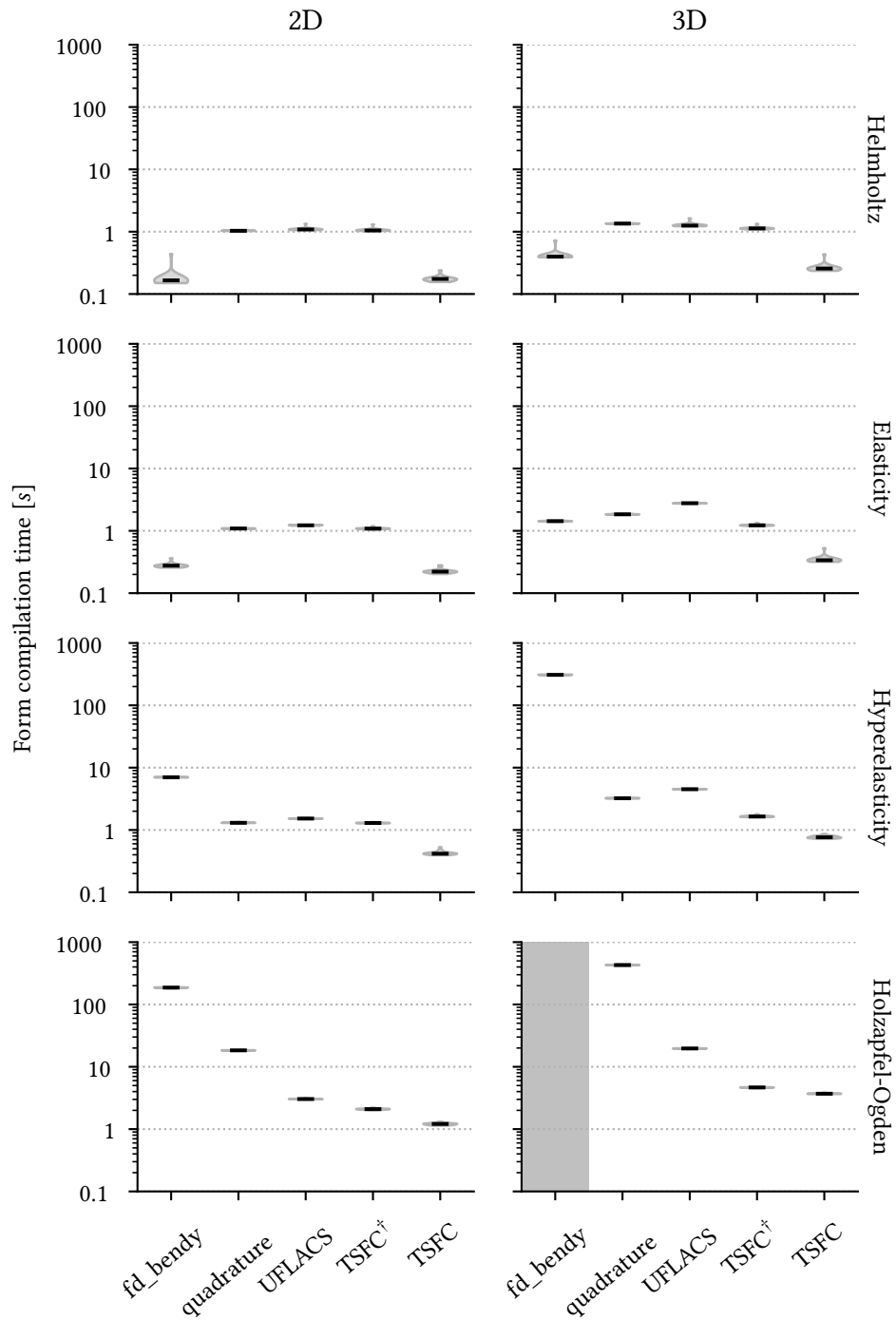


Figure 8.1: Form compilation time for forms of increasing complexity with the following form compilers: FFC quadrature representation, its modified version which supports non-affine cells (`fd_bendy`), UFLACS, TSFC, and TSFC as an FFC representation (`TSFC†`). 3D Holzapfel–Ogden data for `fd_bendy` are missing because compilation ran out of memory after more than 6 hours on a machine with 64GB of memory.

To make the measurements as fair as possible, we configured FFC representations to skip element compilation, and for the C++ compilation we only kept the relevant `tabulate_tensor` method from the generated UFC class. However, since we did not want to change the generated code, we had to include the relevant header files. FFC representations generate code that use functions from the standard C++ header `<algorithm>`, the compilation of which takes about 0.9 seconds on the test machine. Form compilers targeting Firedrake (`fd_bendy` and TSFC) generate C code, and the inclusion of relevant header files induces much less overhead. This effect dominates the simplest test case, the Helmholtz equation. For further fairness, we also included TSFC as an FFC representation (shown as TSFC[†]), including the same set of header files as for the other FFC representations.

All form compilers compile the Helmholtz equation quickly and perform tolerably on the elastic model. The modified version of FFC quadrature with non-affine support (`fd_bendy`) demonstrates excessive compilation times on the simple hyperelastic model already. While non-affine support adds a huge toll to compile-time performance, even the base version of FFC quadrature becomes unmanageable as form complexity increases, as the data for Holzapfel–Ogden model demonstrate. UFLACS and TSFC, however, have tolerable compile-time performance in all cases, TSFC being somewhat faster, but typically within the same order of magnitude.

The overall compilation time of the most difficult case (Holzapfel–Ogden 3D) with TSFC can further be reduced from 3.7 to 2.2 seconds by disabling the optimisation described in section 7.4. This configuration, however, did not yield significant benefits for the other test cases as long as C compilation times are also included.

8.3 Algorithmic complexity for high-order FEM

We now experimentally evaluate the performance benefits of the work in chapters 5 and 6, using the spectral mode of TSFC with all FInAT elements. For comparison, we take a FIAT/coffee mode as baseline:

- Most FInAT elements are disabled, and instead we use the corresponding FIAT elements through the generic wrapper (section 5.1). Vector and tensor elements (section 5.2) are an exception: since we simplified the kernel interface with the FInAT transition, these elements have no direct equivalent in FIAT.
- The transformations of the coffee mode are applied. Kronecker delta nodes are

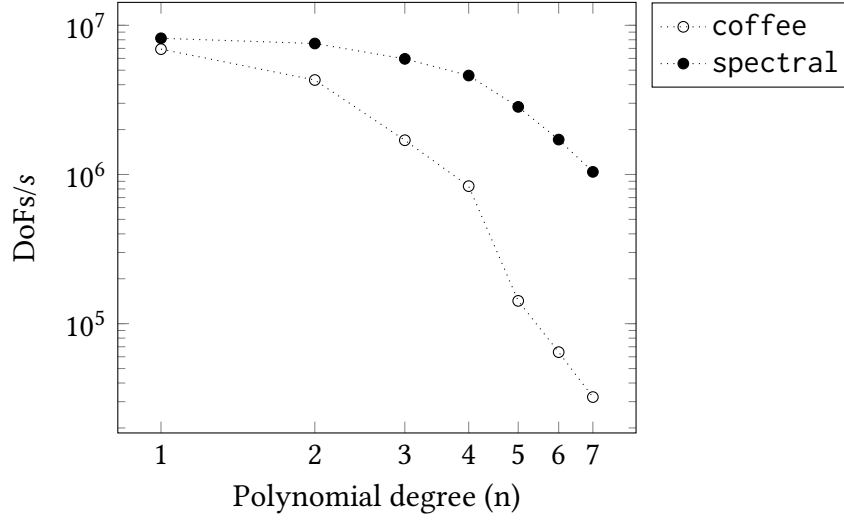


Figure 8.2: Stokes momentum term assembly on a tetrahedral mesh, excluding overheads like sparsity pattern creation and matrix initialisation.

replaced with indexing of an identity matrix.

We consider a range of polynomial degrees, and in most cases we plot the ratio of the number of degrees of freedom (DoFs) and execution time. This metric helps to compare the relative cost of different polynomial degrees.

The experiments were run on a workstation with two 2.6 GHz, 8-core E5-2640 v3 (Haswell) CPUs, for a total of 16 cores. According to its specifications, this CPU has a memory bandwidth of 59 GB/s. The generated C kernels were compiled with `-march=native -O3 -ffast-math` using GCC 5.4.0 provided by Ubuntu 16.04.3 LTS.

8.3.1 Matrix assembly

To demonstrate delta cancellation, we first consider the Stokes momentum term. To rule out the additional effects of sum factorisation, we evaluate this form on a tetrahedral mesh. Fig. 8.2 shows the difference in performance across a range of polynomial degrees, with more speed up for higher degrees. Note that for matrix assembly, if assembling an $N \times N$ sparse matrix takes t seconds, then the DoFs/s rate is N/t .

Since some degrees of freedom are shared between cells, assembly needs to work on those DoFs multiple times. This effect reduces the DoFs/s rate, and is stronger for low polynomial degrees. To approximately quantify this, we consider Q_n elements: the number of unique degrees of freedom – neglecting boundary effects – is exactly n^d per

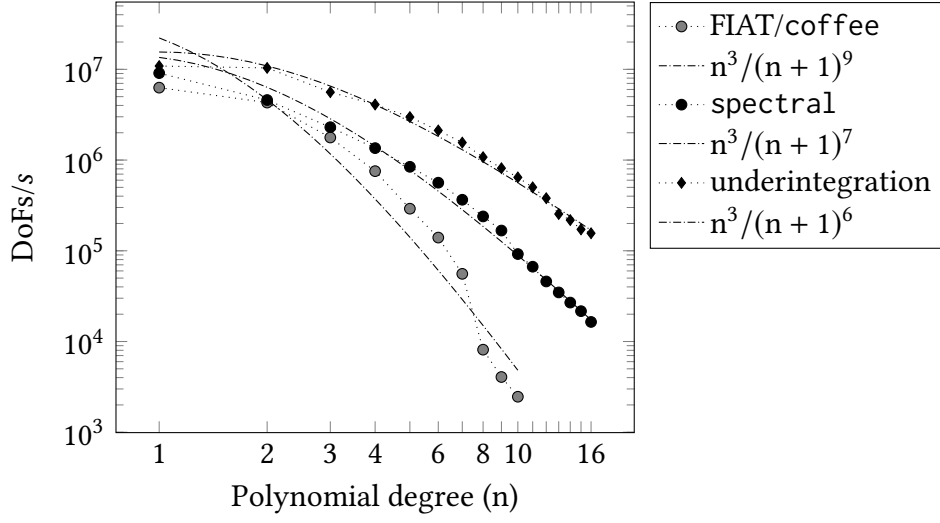


Figure 8.3: Number of degrees of freedom calculated per second for hexahedral Laplace operator *assembly*, excluding overheads like sparsity pattern creation and matrix initialisation.

cell, while the number of degrees of freedom is $(n+1)^d$ for each cell. Suppose $O(n^r)$ is the required number of floating-point operations per cell, where r is the appropriate power. For low polynomial degrees, $c \cdot (n+1)^r$ with some constant c is generally a closer approximation than $c \cdot n^r$. Therefore, assuming a degree-independent FLOPS rate, the DoFs/s measure is approximated as

$$C \cdot n^d / (n+1)^r. \quad (8.24)$$

Next, we consider the Laplace operator on a hexahedral mesh to demonstrate sum factorisation. The expected per-cell algorithmic complexity is $O(n^9)$ without and $O(n^7)$ with sum factorisation, as anticipated in Table 6.1. Fig. 8.3 shows measurement data and confirms these expectations. We see orders of magnitude difference in performance for the highest degrees.

We now demonstrate an interesting combination of delta cancellation and sum factorisation. Still considering the Laplace operator, we take the test and trial functions from a Gauss–Lobatto–Legendre finite element basis, and we consider a quadrature rule with collocated quadrature points (section 5.4). This quadrature rule is insufficient for exact integration, so the user has to specify the quadrature rule manually to enable this optimisation. This case appears in Fig. 8.3 as “underintegration”. However, the

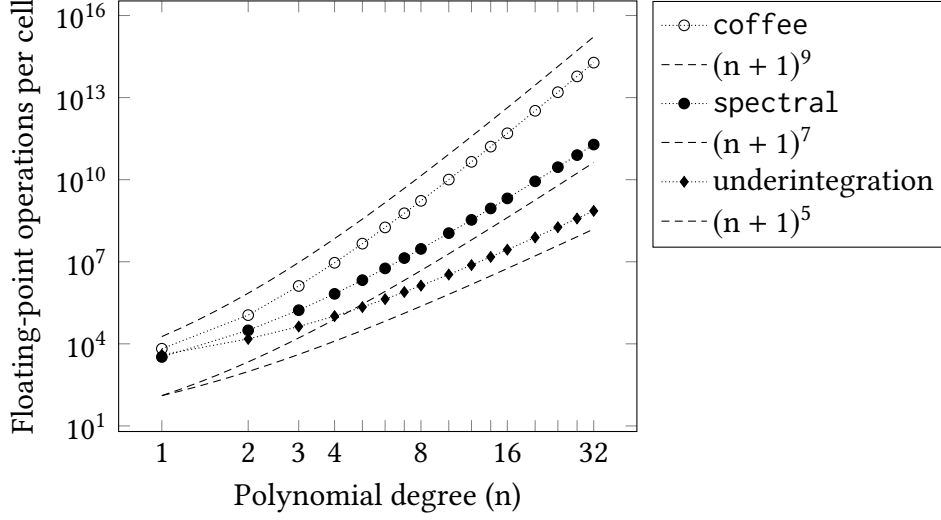


Figure 8.4: Floating-point operations in generated kernels for hexahedral Laplace operator assembly.

collocation enables FInAT to construct the substitution

$$\frac{\partial \Psi_i}{\partial \hat{x}_k}(\xi_q) \mapsto \begin{bmatrix} D\Psi_{i_1 q_1}^{(1)} \delta_{i_2 q_2} \delta_{i_3 q_3} \\ \delta_{i_1 q_1} D\Psi_{i_2 q_2}^{(2)} \delta_{i_3 q_3} \\ \delta_{i_1 q_1} \delta_{i_2 q_2} D\Psi_{i_3 q_3}^{(3)} \end{bmatrix}_k \quad (8.25)$$

where $i = (i_1, i_2, i_3)$ and $q = (q_1, q_2, q_3)$, and $D\Psi^{(1)}$, $D\Psi^{(2)}$, and $D\Psi^{(3)}$ are tabulation matrices for the derivative of the interval elements. The algorithmic complexity of assembling the Laplace operator on nonaffine d -cubes could thus be reduced to $O(n^{d+2})$, while sum factorisation alone gets $O(n^{2d+1})$. Therefore, we would expect “underintegration” to approach $O(n^5)$ on a hexahedral mesh, but in Fig. 8.3 it approaches $O(n^6)$ instead, since *global assembly* in Firedrake still assumes a *dense* element stiffness matrix. However, analytically calculating the number of floating-point operations in the generated kernels, we can confirm that FInAT and TSFC optimise this case correctly (see Fig. 8.4).

Of course, the objective of this work is not simply to have another attempt at optimising the Laplace operator, but to have an automatic code generation system that can carry out these optimisations in principle on any form. As a more complicated example, we consider the the Saint Venant–Kirchhoff hyperelastic model. Fig. 8.5 plots the performance as a function of polynomial degree.

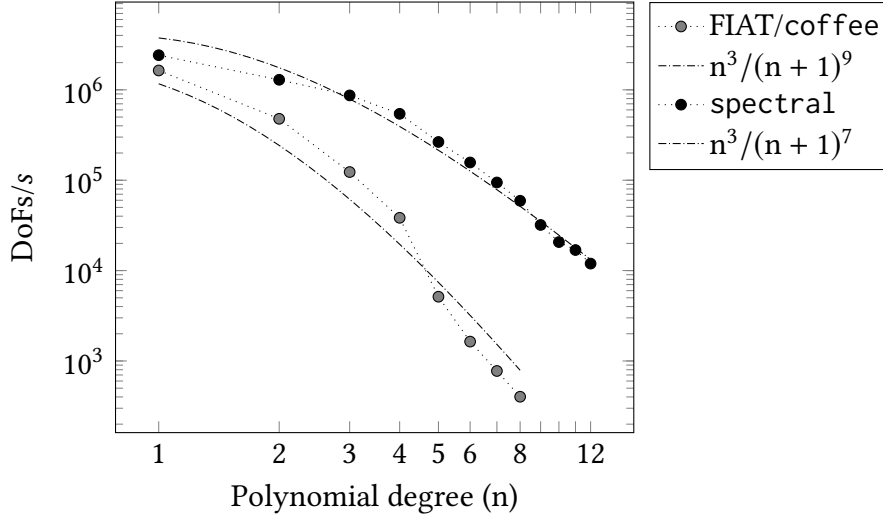


Figure 8.5: Number of degrees of freedom calculated per second for the left-hand side *assembly* of a hexahedral hyperelastic model, excluding overheads like sparsity pattern creation and matrix initialisation.

Finally, we also consider the curl-curl operator to demonstrate sum factorisation on $H(\text{div})$ and $H(\text{curl})$ conforming elements. Here we use NCE elements, the hexahedral $H(\text{curl})$ conforming element in the Q^- family. Fig. 8.6 shows that sum factorisation also works for this finite element with the spectral mode.

One can observe a significant performance drop from degree 7 to 8 when using FIAT/coffee in Fig. 8.3. Although the number of degrees of freedom per direction is a multiple of the vector lane length for degree 7 and not for degree 8, this is likely not merely a vectorisation effect, since there is also performance drop from degree 4 to 5 in Fig. 8.5, and similarly even smaller drops between certain degrees for the other configurations. In most cases, they do not coincide with vector lane length. We have not investigated the actual reason, but one possible explanation could be cache effects: since working set size grows with increasing polynomial degree, at certain points the utilisation of slower cache levels may be necessary.

8.3.2 Operator action and linear forms

Since there are $O(n^d)$ nonzero entries per row in the assembled matrix, n being the polynomial order and d the dimension, the matrix-free application of operator action is especially attractive for higher-order discretisations. The reasons are twofold. First, the

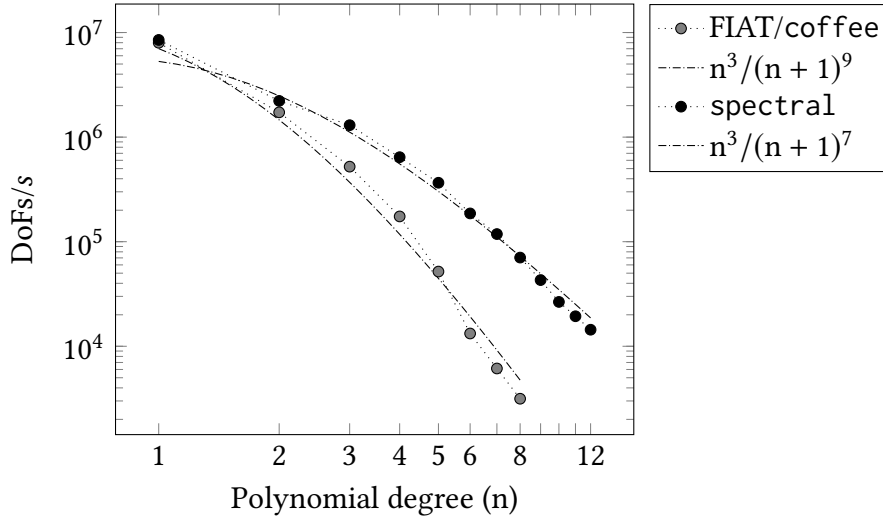


Figure 8.6: Number of degrees of freedom calculated per second for hexahedral curl-curl operator *assembly*, excluding overheads like sparsity pattern creation and matrix initialisation.

memory requirement of storing the assembled matrix may be prohibitively expensive. In fact, this is the reason we only tested matrix assembly for degrees up to 16. At that point, the size of the dense local tensor is 193MB, and we often need several temporaries of similar size, all of which are allocated on the stack. Even with unlimited stack size and just one cell plus halo region per core, we quickly run out of the available 64GB memory. Second, fast finite element assembly with sum factorisation will outperform matrix-vector multiplication with a pre-assembled sparse matrix, even if the cost of matrix assembly is ignored. This is because sparse matrix-vector multiplication takes $O(n^{2d})$ time per cell, while matrix-free assembly of operator action with sum factorisation only needs $O(n^{d+1})$.

We can directly compare sparse matrix-vector multiplication with various approaches to matrix-free algorithms using the metric of degrees of freedom per second, or DoFs/s. This is plotted for three configurations for calculating the action of the Laplace operator in Fig. 8.7, the left-hand side of a hyperelastic model in Fig. 8.8, and the curl-curl operator in Fig. 8.9.

Kirby and Mitchell [2017, §5.1] perform a similar comparison of matrix-free actions to assembled PETSc matrices, but without access to sum factorisation. They find that:

1. Matrix-free applications are generally an $O(1)$ factor slower than matrix-vector

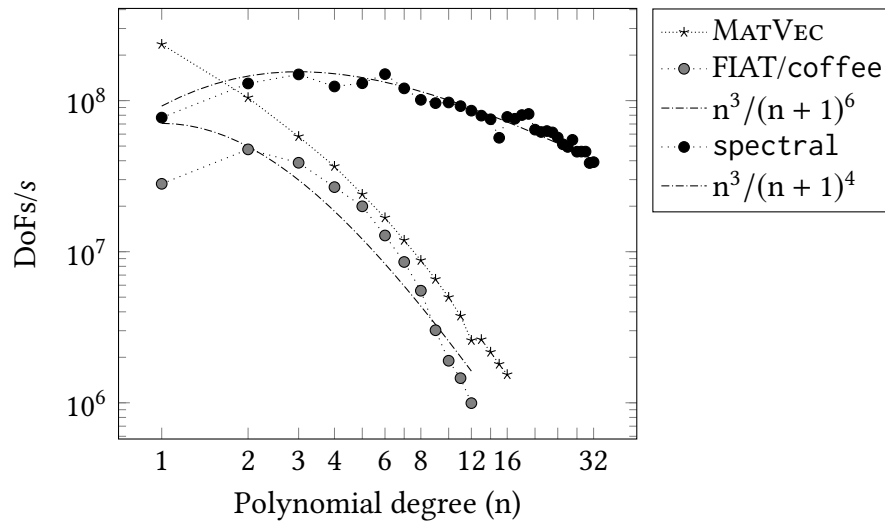


Figure 8.7: Number of degrees of freedom calculated per second for hexahedral Laplace operator *action*.

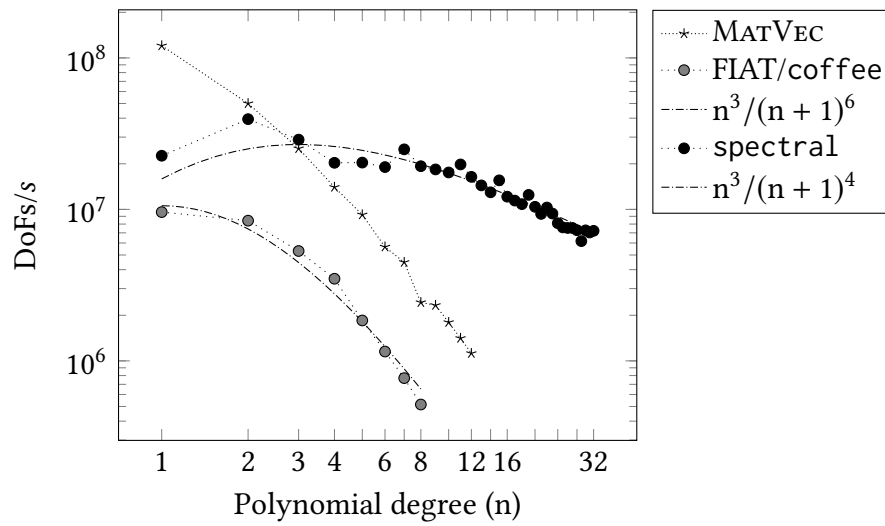


Figure 8.8: Number of degrees of freedom calculated per second for the left-hand side *action* of a hexahedral hyperelastic model.

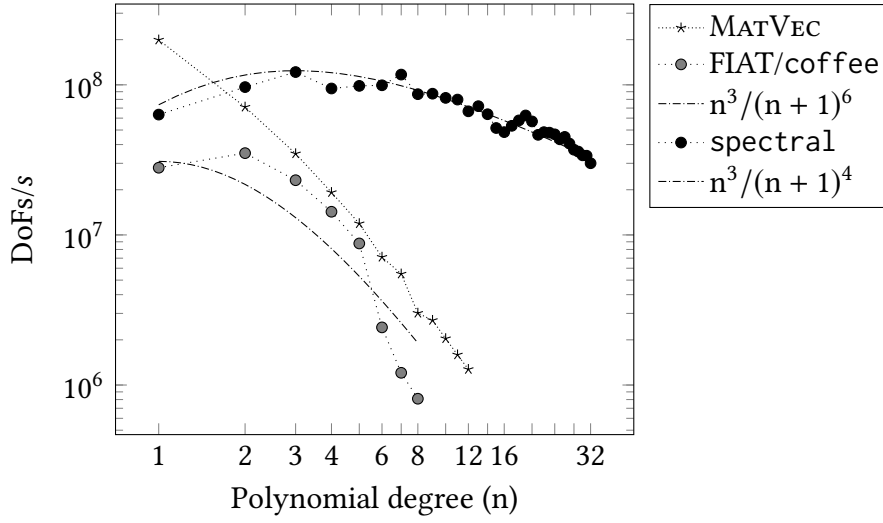


Figure 8.9: Number of degrees of freedom calculated per second for hexahedral curl-curl operator *action*.

products. When the Krylov subspace method runs no more than a few iterations, and the assembled matrix is not otherwise needed, then matrix-free applications could be overall cheaper since the need for costly matrix assembly is eliminated.

2. For high polynomial orders, the memory requirement of the assembled matrix may simply be prohibitive.

Using sum factorised assembly, however, matrix-free applications are the clear fastest choice for high enough polynomial orders. For low orders, one must fall back to the considerations of point 1.

Note that $n^3/(n+1)^4$ initially increases before approaching $O(n^{-1})$ as $n \rightarrow \infty$, so the cost of sum factorised matrix-free application initially decreases with increasing polynomial order. The same applies for $n^2/(n+1)^3$ in 2D. As previously discussed, this is due to the decreasing portion of degrees of freedom that belong to multiple cells. At the same time, this effect renders sparse matrix-vector multiplication especially efficient at low polynomial orders, since the contributions of different cells are aggregated during matrix assembly.

We also observe that coffee mode does not outperform spectral mode even in case of low polynomial orders. This makes the latter a good default choice in TSFC for all cases. Furthermore, the use of FIAT elements was found to be infeasible in the high-order regime, because it resulted in kernels that are over 100MB both as C code

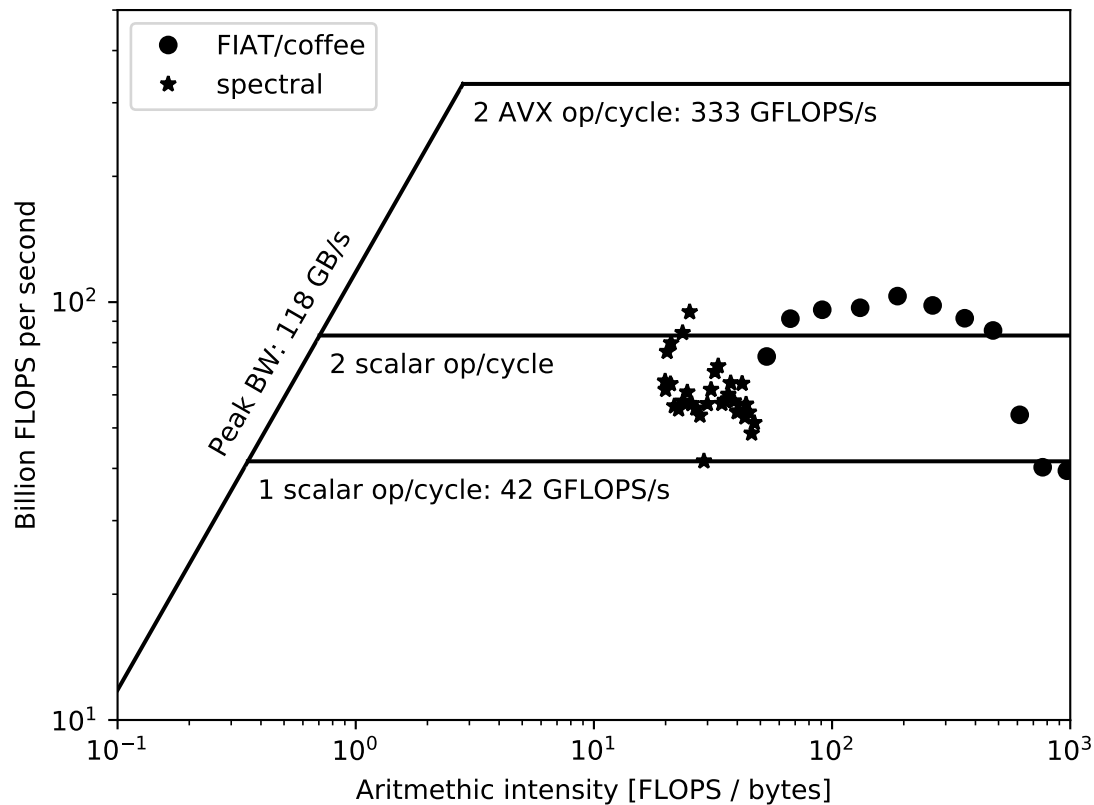


Figure 8.10: Roofline plot for hexahedral Laplace operator *action*, compare with Fig. 8.7. The amount of data to be transferred is estimated assuming perfect reuse. The number of floating-point operations is counted based on the generated C code; the actual number of executed operations may differ due to optimisations of the C compiler.

and as executable binaries. Most of that size is occupied by tabulation matrices, which grow $O(n^{2d})$ in size, nevertheless GCC may not finish compiling them within an hour.

Finally, a note on form compilation time. Argument factorisation has a noticeable, but not prohibitive cost, while all other described transformations are cheap. The Holzapfel–Ogden model has been used for stress testing form compilers, since practical finite element models are seldom more difficult to compile. TSFC compiles its left-hand side in 1.4 seconds in vanilla mode, 5.6 seconds in coffee mode, and 5.9 seconds in spectral mode on the test machine.

A note on hardware utilisation

To gain an insight into the performance of the generated code with respect to hardware limits, we present the roofline plot for the hexahedral Laplace operator action in Fig. 8.10. The presented data are characterised by high arithmetic intensity and low utilisation of the vector capabilities of the CPU. Although Kirby and Mitchell [2017, Fig. 3] measure significantly lower arithmetic intensity with Firedrake, they use a tetrahedral mesh rather than a hexahedral mesh. This implies several differences:

- Tetrahedral meshes are fully unstructured in Firedrake, while hexahedral meshes are only supported in Firedrake as extruded quadrilateral meshes. This structured dimension means that indirection maps are required only for the 2-dimensional base mesh, which reduces the amount of data to be transferred by a significant fraction, thereby increasing arithmetic intensity.
- Flat tetrahedra have an affine geometric transformation, while hexahedra are generally non-affine. This means that the Jacobian, its inverse and determinant are to be evaluated at each quadrature point for hexahedra, while they are constant within each tetrahedron. For a simple form such as the Laplace operator, this represents a significant overhead.

We also note that the FIAT/coffee mode has noticeably higher arithmetic intensity than the spectral mode, even at lowest order. This is most likely due to the optimisation of vector elements (section 6.2), which are used for the representation of the coordinate field, e.g., when evaluating the Jacobian of the coordinate transformation.

Based on the data shown in Fig. 8.10, we may draw the following conclusions:

- The computational kernels are not in the bandwidth limited regime, which justifies our approach of optimising the number of operations. We can see that the reduced arithmetic intensity directly translates to shorter execution time.
- The vector capabilities of the CPU appear poorly utilised with only circa 1–2 operations per cycle. Kronbichler and Kormann [2017] demonstrate much better utilisation of modern hardware, as well as Müthing et al. [2017], although the latter solve a different equation. This suggests a direction for future research and a potential for further improvement in code generation.

Chapter 9

Summary & outlook

9.1 Summary

This thesis represents a set of significant contributions to code generation techniques for the finite element method. The *Two-Stage Form Compiler* (TSFC) has been developed with a vigorous commitment to viewing a compiler as a series of *compiler passes* (transformations) over layers of *intermediate languages*. This is in contrast with templating and string-pasting techniques, often employed – at least in part – by previous form compilers, which result in poor composability and render optimisations very difficult.

GEM, the intermediate representation of TSFC, creates a clean separation between the implementation of finite element objects and geometric terms (chapter 4) as well as the problem of efficient code generation for tensor algebra expressions (chapter 7). Furthermore, GEM is proven to be a suitable medium and level of abstraction for numerous optimisations, including the expression and exploitation of structure intrinsic to some finite elements (chapters 5 and 6). Another contribution of TSFC is the simple, yet novel method for separating the contributions of the form in discontinuous Galerkin methods (section 4.2).

TSFC with FInAT can generate finite assembly code with optimal algorithmic complexity for high-order tensor product elements. Since tensor product elements naturally require tensor product cells such as quadrilateral and hexahedra, I had to ensure that the rest of Firedrake supports these cell types as well. I have thus proposed a distributed-memory parallel algorithm, extending the serial algorithm in deal.II, for consistent quadrilateral edge orientations. I provided a proof that there are consistent edge orientations for any quadrilateral mesh of an orientable 2-dimensional manifold.

My novel analysis establishes a link between the well-known *Union-Find* algorithm and assigning orientations to the edges of a quadrilateral mesh, thus existing parallelisation approaches for the Union-Find algorithm can be easily adapted to construct further parallel algorithms for mesh orientations.

The practical implications on numerical methods research are also important. The efficient traversal of expressions keeps the compilation time with TSFC acceptably low even for highly complicated forms. This is crucial for interactive use by those developing complicated numerical solvers, and it was a crucial requirement for properly supporting non-simplicial cells as well as higher-order mesh geometry in Firedrake.

With FInAT and TSFC, a numerical researcher can just write the weak form in UFL, and automatically get: a) sum factorisation with continuous, discontinuous, $H(\text{div})$ and $H(\text{curl})$ conforming elements on cuboid cells; b) optimised evaluation at collocated quadrature points when underintegration is requested; and c) minor optimisations with vector and tensor elements. These techniques were known and have been applied in hand-written numerical software before, however, we are now able to utilise these optimisations in an automatic code generation setting in Firedrake. This is especially useful in combination with matrix-free methods [Kirby and Mitchell, 2017], enabling the development of efficient high-order numerical schemes while retaining high productivity. My measurements show that on modern hardware one can increase the polynomial degree up to 8–10 without a notable increase in the run time of matrix-free operator applications (per degree of freedom).

I would like to especially emphasise sum factorisation with $H(\text{div})$ or $H(\text{curl})$ conforming elements. We only know of Hientzsch [2001] implementing such a scheme, and his implementation was manually written. Writing such finite element assembly routines aggregates several layers of complexity: a) the combination of Piola mappings with non-affine mesh geometry (implied by non-simplicial cells) makes changing to reference space a quite complicated operation; b) the reference form has to be split along the *sum element* to recover the product structure of the subelements; c) tensor contractions has to be rearranged for sum factorisation; and d) one must produce low-level code for the sum factorised expressions. Referring back to the *thesis statement*: code generation allows the decomposition of these layers of complexity such that they can be solved and tested as individual, smaller problems.

9.2 Future work

Finally, I give a few suggestions and ideas for future work that is related to and expands upon the work presented in this thesis.

9.2.1 Sum factorisation on simplicial cells

I have explained the generation of high-order finite element kernels with optimal algorithmic complexity on tensor product cells such as quadrilaterals and hexahedra. The lack of support for fast finite element assembly on simplicial cells is less than ideal for various reasons. For example, meshing a complicated geometry is significantly less difficult with simplices, especially in 3D.

Kirby and Thinh [2012] describe the theory of producing fast finite element operators on simplicial elements with Bernstein polynomials. They achieve the same algorithmic complexity on d -simplices that we have demonstrated on d -cubes. Kirby [2014b] constructs simplicial $H(\text{div})$ and $H(\text{curl})$ conforming elements using Bernstein polynomials that are amenable to transformations that results in similar low-complexity operators. Ainsworth et al. [2011] further reduce the complexity of matrix assembly to $O(n^{2d})$, which implies constant work per degree of freedom.

A key requirement for supporting all this is the extension of GEM and its code generation to support *triangular shapes*. Further steps include the implementation of Bernstein elements in FInAT, and suitable quadrature rules that yield semi-separable tabulation expressions on Bernstein elements. The existing sum factorisation algorithm, however, would likely be suitable without much change.

9.2.2 Low-level performance on modern hardware

While section 8.3 confirms optimal algorithmic complexity for high-order operators as a function of polynomial degree, I did not consider the constant factors hidden in the big O notation: neither compared to other, manually written spectral codes, nor against the absolute performance limits of the hardware. This evaluation and – most likely – the optimisation of low-level performance remain to be done.

For this purpose, it may be worthwhile to consider Loo.py [Klöckner, 2014, 2015], which has a representation that is slightly lower level than GEM, and offers various transformations that facilitate *vectorisation* as well as code generation for GPUs. For example, *cross-element* vectorisation is fairly straightforward if the local kernel and the

global element loop are represented as one joint `Loo.py` kernel. Recent manuscripts by [Kronbichler and Kormann \[2017\]](#) and [Müthing et al. \[2017\]](#) furthermore describe the optimisations that they used to achieve good utilisation of modern CPUs. Further considerations for GPUs include the *thread transposition* approach [[Knepley et al., 2016](#)] and the related work in section 6.1 about reducing the *working set size*.

9.2.3 Transferring remaining COFFEE functionality

COFFEE carries out both low-level optimisations, such as alignment and padding to facilitate vectorisation by the host C compiler, and high-level optimisations to reduce the operation count which require restructuring the loops and expressions. COFFEE seeks to determine the optimal trade-off amongst common subexpressions elimination, loop-invariant code motion, and vectorisation. Moreover, its cost model includes the working set size of the kernel which is thus taken into account when deciding about transformations that reduce the number of operations.

Tianjiao Sun has re-implemented a simplified version of the COFFEE algorithm to operate on GEM instead. The simplifications include: a) unconditional argument factorisation; and b) disregarding the memory effects of minimising the number of operations. Furthermore, this technology transfer so far did not extend to the low-level optimisations implemented in COFFEE. Since COFFEE is being retired, the transfer of these features remain future work. Amongst them, the consideration of working set size may become especially important when targeting GPUs.

9.2.4 Global assembly of sparse local tensors

We have seen earlier that vector elements (section 5.2) and collocated quadrature points (section 5.4) cause particular nonzero patterns in the local tensor to emerge (section 6.5). However, as explained on page 105 (compare Figs. 8.3 and 8.4), we cannot make full use of this, because the global assembler assumes that the local tensor is always dense.

To overcome this limitation, one must rethink the interface between the global assembler and the form compiler. The form compiler shall communicate – along with each kernel – any nonzero patterns statically recognised in the assembled local tensor. Then the code generation for global assembly shall be extended to avoid allocating space for as well as inserting entries foreknown to be zero.

9.2.5 Countering beta expansion

We have seen in section 7.1 that the elimination of ComponentTensor nodes simplifies the intermediate representation, but it can in certain cases lead to code duplication. This is clearly visible in Algorithm 14 where lines 20 and 27 compute the same values with different free indices, because one is used for the trial function, while the other is used for the test function. Though in this particular example the duplicate computation disappears once geometric loops are unrolled (section 7.4, see Algorithm 17), it remains in other cases. For example, the ILP factorisation algorithm [Luporini et al., 2017] lifts out the same subexpressions from the test and trial function expressions in symmetric forms such as the Laplace operator $\int \nabla u \cdot \nabla v \, dx$, but they are seen as different due to different free indices. This effect is even more apparent with collocated quadrature points (section 5.2), which result in the substitution of quadrature indices with basis function indices when the particular nonzero patterns are recognised (section 6.5).

One might think of avoiding the step in section 7.1 altogether, and implementing code generation for ComponentTensors. However, the high-level constructs described at the end of section 4.3 as well as some internal processing in TSFC and FInAT often create layers of indexed component tensors whose elimination is actually very beneficial.

A smarter approach could be the reconstruction of expressions in form that we may call *index normal form*. This requires a series of *canonical index* objects $(\iota_1, \iota_2, \iota_3, \dots)$. Each subexpression is to be mapped to have the first few canonical indices as its free indices. For example, $a_{jk} * b_{ij}$ has free indices i, j, k , and it is mapped to $a_{\iota_1 \iota_2} * b_{\iota_3 \iota_1}$ which has free indices $\iota_1, \iota_2, \iota_3$ instead. However, the subexpression $b_{\iota_3 \iota_1}$ is not in normal form, which ought to be $b_{\iota_1 \iota_2}$. Therefore, we shall introduce an *index substitution*:

$$a_{\iota_1 \iota_2} * b_{\iota_1 \iota_2} \Big|_{\substack{\iota_1 \rightarrow \iota_3 \\ \iota_2 \rightarrow \iota_1}} \quad (9.1)$$

So, in index normal form, each subexpression (except for any index substitution nodes) has the first few canonical indices as its free indices. If the assignment of canonical indices $(\iota_1, \iota_2, \iota_3)$ to source indices (i, j, k) is calculated deterministically based on the construction of the expression, then all source subexpressions that only differ in the identity of some free indices are mapped to the same expression in the index normal form. Although the construction of this normal form is eager to insert index substitutions like in eq. (9.1), it is fairly straightforward to distinguish subexpressions that are genuinely used with different indices, from those which are only used through one particular index substitution. In the latter case, the extra nodes are easily eliminated by rewriting the subexpression with the substituted indices.

One may say that carrying out the beta expansion step and then later undoing its harmful effects is a wasteful approach. However, this need not be done since the presence of `ComponentTensor` nodes in the source expression does not make the construction of the index normal form much more difficult. Nevertheless, the described approach should be capable of undoing unnecessary beta expansion, and it might also replace the *index caching* presently in use for coefficient evaluations in TSFC.

In any case, to be able to index the same generated temporary with logically different indices will require changes to the current code generation. Generating `Loo.py`, as suggested for consideration in section 9.2.2, might also help with this particular issue.

9.2.6 Sum factorised evaluation of *future* facet integrals

Although the techniques presented in this thesis achieve as much sum factorisation on currently supported facet integrals as one may expect, some issues specific to facet integrals are bound to come up in the future. To explain these issues, I take coefficient evaluation as an example. Let u be a coefficient, and let M be the tabulation matrix of its finite element at the quadrature points. Then $\bar{u}_q := u(F_K(\xi_q))$ is evaluated as

$$\bar{u}_q = \sum_r M_{qr} \mathbf{u}_r \quad (9.2)$$

where $\mathbf{u}$ is the coefficient vector. For facet integrals, the quadrature rule is constructed on the reference cell of the facet, and then mapped to each facet of the reference cell of physical cell. This gives the tabulation matrices $M_1, M_2, \dots, M_N$ where N is the number of facets. Forming a single rank-three tensor M , the evaluation rule becomes

$$\bar{u}_q = \sum_r M_{fqr} \mathbf{u}_r \quad (9.3)$$

where f is the *facet number*. f is not a free index, but a `VariableIndex`: a fixed index whose value is only known at run time (section 4.6).

Tensor product elements with tensor product quadrature rules produce tabulation expressions which are Kronecker products of smaller tabulation matrices. For example, the naïve evaluation rule for a hexahedral cell integral would look like

$$\bar{u}_{(q_1, q_2, q_3)} = \sum_{r_1, r_2, r_3} M_{q_1 r_1}^{(1)} M_{q_2 r_2}^{(2)} M_{q_3 r_3}^{(3)} \mathbf{u}_{(r_1, r_2, r_3)}. \quad (9.4)$$

Notice how each 1-dimensional tabulation matrix couples a quadrature index and a basis function index. If we were to integrate on the facet of a hexahedron, then we still

had three basis function indices (r_1, r_2, r_3), but only two quadrature indices (e.g., q_1 and q_2). The tabulation expression would still have the product structure for each particular facet, for example

$$M_{q_1 r_1}^{(1)} M_{q_2 r_2}^{(2)} M_{r_3}^{(3)}.$$

However, now the coupling between quadrature and basis function indices is different for different facets, which is currently quite problematic when trying to merge tabulation expressions to feature higher ranked tensors indexed with the facet number.

Notice that the example above is for a facet integral on an *unstructured* hexahedral mesh. Firedrake does not presently support this type of mesh, and for supported mesh types this issue does not come up, since

- all non-extruded cell types (interval, triangle, quadrilateral, tetrahedron) have simplex facets (point, interval, triangle) which express no product structure; and
- extruded meshes separate the horizontal and vertical directions with distinct facet integral types.

Consequently, this issue would come up if Firedrake supported unstructured hexahedral meshes, or if sum factorisation on triangles were supported (section 9.2.1).

Further challenges include sum factorised evaluation on trace elements, especially continuous traces for static condensation. The difficulty is that the degrees of freedom (DoFs) of the element cannot be laid out in a continuous array such that the DoFs of each facet can be indexed as a strided dense matrix. This means that a level of indirection for accessing local DoFs is probably necessary to achieve sum factorisation.

9.2.7 Finite elements with some continuity of derivatives

In section 2.2 I told how FFC initially only supported continuous elements, and was later extended with discontinuous, and then with $H(\text{div})$ and $H(\text{curl})$ conforming elements. The latter require contravariant and covariant Piola mappings when changing from physical to reference space (section 2.1.2). Hermite [Ciarlet and Raviart, 1972], Morley [1971], Argyris [Argyris et al., 1968], and Bell [1969] are scalar-valued finite elements with stronger than C^0 continuity. They are currently not supported in either FEniCS or Firedrake. Unlike currently supported elements, which require the same *reference value transformation* for all their basis functions, these elements need different transformations for different basis functions. Kirby [2017] has worked out a general approach to transforming finite elements between physical and reference space, with

the purpose of supporting these elements in an automated code generation framework such as FEniCS and Firedrake.

The core issue here is that the correct reference transformation requires both knowledge that is intrinsic to the finite element, and knowledge of the physical geometry characterised by the coordinate element and the $\mathbf{x}^K$ coefficient vector as defined in section 2.1.3. A possible solution is that FInAT defines an abstract interface through which GEM expressions for various physical geometric quantities may be obtained. TSFC knows the physical geometry during form compilation, so it could provide a concrete implementation of this abstract interface when requesting tabulations from FInAT elements. This enables FInAT to “call back” to TSFC for what it needs to construct the correct reference transformation as described in Kirby [2017].

9.2.8 Dual evaluation in FInAT

FIAT offers a complete implementation for most of its elements: basis evaluation, dual evaluation through either quadrature rules or *Riesz representation*, and most simplicial elements feature a nodal basis. Unfortunately, this is not true for FInAT yet, which thus far has only offered basis evaluation functionality. While we still have access to FIAT elements, and they are used to generate *interpolation* kernels, the lack of structure in the dual evaluation rules prevents the generation of sum factorised interpolation kernels, which would be important, e.g., for uniformly refined geometric multigrid.

9.2.9 Generating GEM from SLATE?

SLATE [Gibson et al., 2017] is a language for local linear algebra operations on the results of finite element assembly kernels. It is used to facilitate hybridisation and static condensation. SLATE leaf nodes typically contain UFL forms, while SLATE interior nodes express linear algebra operations over the local tensors that arise from assembling those forms. In outline, SLATE works as follows:

- (1) The UFL forms in SLATE leaf nodes are passed to the form compiler which produces C functions that carry out local assembly.
- (2) The SLATE compiler translates the SLATE expression. Leaf nodes are translated as function calls to the TSFC-generated kernels. Interior nodes are translated as C++ expressions that make use of Eigen [Guennebaud et al., 2010].

- (3) Global assembly inserts the result of the local SLATE kernel into the appropriate global data structures.

Currently, the SLATE compiler generates C++ directly. It may be worthwhile to consider generating GEM instead. Although this approach requires adding a couple of new node types to GEM such as matrix inversion and determinant, it offers several advantages:

- GEM is a higher-level representation than C++, therefore it ought to be easier to generate as a target language.
- Since the finite element assembly kernels are also represented in GEM at some point, the proper fusion of different assembly kernels as well as the overall SLATE kernel would be much easier to achieve.
- Improvements to code generation from GEM – like vectorisation, other low-level optimisations, and targetting GPUs – would more or less automatically apply to SLATE as well.

Bibliography

- Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, Berkeley, CA, USA, 265–283. arXiv:1605.08695 <https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf>
- Rainer Agelek, Michael Anderson, Wolfgang Bangerth, and William L. Barth. 2017. On Orienting Edges of Unstructured Two- and Three-Dimensional Meshes. *ACM Trans. Math. Software* 44, 1, Article 5 (July 2017), 22 pages. <https://doi.org/10.1145/3061708> arXiv:1512.02137
- Mark Ainsworth, Gaele Andriamaro, and Oleg Davydov. 2011. Bernstein–Bézier Finite Elements of Arbitrary Order and Optimal Assembly Procedures. *SIAM Journal on Scientific Computing* 33, 6 (2011), 3087–3109. <https://doi.org/10.1137/11082539X>
- A. Allam, J. Ramanujam, Gerald Baumgartner, and P. Sadayappan. 2006. Memory Minimization for Tensor Contractions Using Integer Linear Programming. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing (IPDPS’06)*. IEEE Computer Society, Washington, DC, USA, 8. <https://doi.org/10.1109/IPDPS.2006.1639717>
- Martin Sandve Alnæs. 2016. UFL Analyser and Compiler System (UFLACS). (2016). Retrieved 27 November 2017 from <https://bitbucket.org/fenics-project/uflacs-deprecated>
- Martin Sandve Alnæs, Jan Blechta, Johan Hake, August Johansson, Benjamin Kehlet,

- Anders Logg, Chris Richardson, Johannes Ring, Marie E. Rognes, and Garth N. Wells. 2015. The FEniCS Project Version 1.5. *Archive of Numerical Software* 3, 100 (2015), 9–23. <https://doi.org/10.11588/ans.2015.100.20553>
- Martin Sandve Alnæs, Anders Logg, Kristian B. Ølgaard, Marie E. Rognes, and Garth N. Wells. 2014. Unified Form Language: A Domain-Specific Language for Weak Formulations of Partial Differential Equations. *ACM Trans. Math. Software* 40, 2 (2014), 1–37. <https://doi.org/10.1145/2566630> arXiv:1211.4047
- Martin Sandve Alnæs and Kent-André Mardal. 2010. On the efficiency of symbolic computations combined with code generation for finite element methods. *ACM Trans. Math. Software* 37, 1 (2010), 6:1–6:26. <https://doi.org/10.1145/1644001.1644007>
- Richard J. Anderson and Heather Woll. 1991. Wait-free Parallel Algorithms for the Union-find Problem. In *Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing (STOC '91)*. ACM, New York, NY, USA, 370–380. <https://doi.org/10.1145/103418.103458>
- J. H. Argyris, I. Fried, and D. W. Scharpf. 1968. The TUBA Family of Plate Elements for the Matrix Displacement Method. *The Aeronautical Journal* 72, 692 (1968), 701–709.
- Douglas N. Arnold, Franco Brezzi, and Michel Fortin. 1984. A stable finite element for the stokes equations. *CALCOLO* 21, 4 (01 Dec 1984), 337–344. <https://doi.org/10.1007/BF02576171>
- Douglas N. Arnold and Anders Logg. 2014. Periodic Table of the Finite Elements. *SIAM News* 47, 9 (November 2014), 212. <https://femtable.org/>
- David A. Bader and Guojing Cong. 2005. A Fast, Parallel Spanning Tree Algorithm for Symmetric Multiprocessors (SMPs). *J. Parallel and Distrib. Comput.* 65, 9 (Sept. 2005), 994–1006. <https://doi.org/10.1016/j.jpdc.2005.03.011>
- Babak Bagheri and Ridgway Scott. 2003. About Analysa. (2003). <http://people.cs.uchicago.edu/~ridg/al/aa.html>
- Gabriel Balaban, Martin S. Alnæs, Joakim Sundnes, and Marie E. Rognes. 2016. Adjoint multi-start-based estimation of cardiac hyperelastic material parameters using shear data. *Biomechanics and Modeling in Mechanobiology* 15, 6 (2016), 1509–1521. <https://doi.org/10.1007/s10237-016-0780-7> arXiv:1603.03796

- Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Dave A. May, Lois Curfman McInnes, Karl Rupp, Patrick Sanan, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. 2017a. *PETSc Users Manual*. Technical Report ANL-95/11 - Revision 3.8. Argonne National Laboratory. <http://www.mcs.anl.gov/petsc>
- Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Dave A. May, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. 2017b. PETSc Web page. <http://www.mcs.anl.gov/petsc>. (2017). <http://www.mcs.anl.gov/petsc>
- Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. 1997. Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. In *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen (Eds.). Birkhäuser Press, 163–202.
- Wolfgang Bangerth, Michael Anderson, and Ralf Hartmann. 2003. The deal.II Library: GridReordering<dim, spacedim> Class Template Reference. (2003). <https://www.dealii.org/developer/doxygen/deal.II/classGridReordering.html>
- Wolfgang Bangerth, Ralf Hartmann, and Guido Kanschat. 2007. Deal.II—A General-purpose Object-oriented Finite Element Library. *ACM Trans. Math. Software* 33, 4, Article 24 (Aug. 2007), 27 pages. <https://doi.org/10.1145/1268776.1268779>
- Peter Bastian, Markus Blatt, Christian Engwer, Andreas Dedner, Sreejith Kuttanikkad, Mario Ohlberger, and Oliver Sander. 2006. The distributed and unified numerics environment (DUNE). In *Proceedings of the 19th Symposium on Simulation Technique in Hannover*, Vol. 123. 6. https://ivv5hpp.uni-muenster.de/u/mohl_b_01/postscript/dune_asim2006.pdf
- Peter Bastian, Christian Engwer, Jorrit Fahlke, Markus Geveler, Dominik Göddeke, Oleg Iliev, Olaf Ippisch, René Milk, Jan Mohring, Steffen Müthing, Mario Ohlberger, Dirk Ribbrock, and Stefan Turek. 2016. *Hardware-Based Efficiency Advances in the EXA-DUNE Project*. Springer International Publishing, Cham, 3–23. https://doi.org/10.1007/978-3-319-40528-5_1

- Peter Bastian, Christian Engwer, Dominik G ddke, Oleg Iliev, Olaf Ippisch, Mario Ohlberger, Stefan Turek, Jorrit Fahlke, Sven Kaulmann, Steffen M thling, and Dirk Ribbrock. 2014. EXA-DUNE: Flexible PDE Solvers, Numerical Methods and Applications. In *Euro-Par 2014: Parallel Processing Workshops (Lecture Notes in Computer Science)*, Lu s Lopes, Julius  zilinskas, Alexandru Costan, Roberto G. Cascella, Gabor Kecskemeti, Emmanuel Jeannot, Mario Cannataro, Laura Ricci, Siegfried Benkner, Salvador Petit, Vittorio Scarano, Jos  Gracia, Sascha Hunold, Stephen L. Scott, Stefan Lankes, Christian Lengauer, Jesus Carretero, Jens Breitbart, and Michael Alexander (Eds.), Vol. 8806. Springer, Berlin, Heidelberg, 530–541. https://doi.org/10.1007/978-3-319-14313-2_45
- Gerald Baumgartner, Alexander A. Auer, David E. Bernholdt, Alina Bibireata, Venkatesh Choppella, Daniel Cociorva, Xiaoyang Gao, Robert J. Harrison, So Hirata, Sriram Krishnamoorthy, Sandhya Krishnan, Chi-Chung Lam, Qingda Lu, Marcel Nooijen, Russell M. Pitzer, J. Ramanujam, P. Sadayappan, and Alexander Sibiryakov. 2005. Synthesis of high-performance parallel programs for a class of *Ab Initio* quantum chemistry models. *Proc. IEEE* 93, 2 (Feb 2005), 276–292. <https://doi.org/10.1109/JPROC.2004.840311>
- Kolbein Bell. 1969. A refined triangular plate bending finite element. *Internat. J. Numer. Methods Engrg.* 1, 1 (1969), 101–122. <https://doi.org/10.1002/nme.1620010108>
- Susanne C. Brenner and L. Ridgway Scott. 2008. *The Mathematical Theory of Finite Element Methods*. Springer.
- Chris D. Cantwell, David Moxey, A. Comerford, A. Bolis, G. Rocco, Gianmarco Mengaldo, Daniele De Grazia, S. Yakovlev, J. E. Lombard, D. Ekelschot, B. Jordi, H. Xu, Y. Mohamied, C. Eskilsson, B. Nelson, P. Vos, C. Biotto, R. M. Kirby, and S. J. Sherwin. 2015. Nektar++: An open-source spectral/hp element framework. *Computer Physics Communications* 192 (2015), 205–219. <https://doi.org/10.1016/j.cpc.2015.02.008>
- P. G. Ciarlet and P. A. Raviart. 1972. General Lagrange and Hermite interpolation in $\mathbb{R}^n$ with applications to finite element methods. *Archive for Rational Mechanics and Analysis* 46, 3 (01 Jan 1972), 177–199. <https://doi.org/10.1007/BF00252458>
- Daniel Cociorva, Gerald Baumgartner, Chi-Chung Lam, P. Sadayappan, J. Ramanujam, Marcel Nooijen, David E. Bernholdt, and Robert Harrison. 2002. Space-

- time Trade-off Optimization for a Class of Electronic Structure Calculations. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*. ACM, New York, NY, USA, 177–186. <https://doi.org/10.1145/512529.512551>
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. *Introduction to Algorithms* (2nd ed.). MIT Press, Cambridge, MA, USA.
- George Cybenko, Tom G. Allen, and J. E. Polito. 1988. Practical parallel Union-Find algorithms for transitive closure and clustering. *International Journal of Parallel Programming* 17, 5 (01 Oct 1988), 403–423. <https://doi.org/10.1007/BF01383882>
- Patrick Dular, Christophe Geuzaine, F. Henrotte, and Willy Legros. 1998. A general environment for the treatment of discrete problems and its application to the finite element method. *IEEE Transactions on Magnetics* 34, 5 (Sep 1998), 3395–3398. <https://doi.org/10.1109/20.717799>
- Evgeny Epifanovsky, Michael Wormit, Tomasz Kuś, Arie Landau, Dmitry Zuev, Kirill Khistyayev, Prashant Manohar, Ilya Kaliman, Andreas Dreuw, and Anna I. Krylov. 2013. New implementation of high-level correlated methods using a general block tensor library for high-performance electronic structure calculations. *Journal of Computational Chemistry* 34, 26 (2013), 2293–2309. <https://doi.org/10.1002/jcc.23377>
- Paul Fischer, J. Kruse, J. Mullen, H. Tufo, J. Lottes, and S. Kerkemeier. 2008. NEK5000: A fast and scalable open-source spectral element solver for CFD. (2008). Retrieved 27 October 2017 from <https://nek5000.mcs.anl.gov>
- Christophe Geuzaine and Jean-François Remacle. 2009. Gmsh: A 3-D finite element mesh generator with built-in pre-and post-processing facilities. *Internat. J. Numer. Methods Engrg.* 79, 11 (2009), 1309–1331. <https://doi.org/10.1002/nme.2579>
- Thomas H. Gibson, Lawrence Mitchell, David A. Ham, and Colin J. Cotter. 2017. A domain-specific language for the hybridization and static condensation of finite element methods. (2017). In preparation.
- Michael B. Giles, Gihan R. Mudalige, Zohirul Sharif, Graham R. Markall, and Paul H. J. Kelly. 2012. Performance Analysis and Optimization of the OP2 Framework on

- Many-Core Architectures. *Comput. J.* 55, 2 (Feb. 2012), 168–180. <https://doi.org/10.1093/comjnl/bxr062>
- Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. (2010). <http://eigen.tuxfamily.org>
- Cyrus Harrison, Hank Childs, and Kelly Gaither. 2011. Data-parallel Mesh Connected Components Labeling and Analysis. In *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization (EGPGV '11)*. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 131–140. <https://doi.org/10.2312/EGPGV/EGPGV11/131-140>
- Albert Hartono, Qingda Lu, Xiaoyang Gao, Sriram Krishnamoorthy, Marcel Nooijen, Gerald Baumgartner, David E. Bernholdt, Venkatesh Choppella, Russell M. Pitzer, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2006. Identifying cost-effective common subexpressions to reduce operation count in tensor contraction evaluations. In *Computational Science – ICCS 2006 (Lecture Notes in Computer Science)*, Vassil N. Alexandrov, Geert Dick van Albada, Peter M. A. Sloot, and Jack Dongarra (Eds.), Vol. 3991. Springer, Berlin, Heidelberg, 267–275. https://doi.org/10.1007/11758501_39
- Albert Hartono, Alexander Sibiryakov, Marcel Nooijen, Gerald Baumgartner, David E. Bernholdt, So Hirata, Chi-Chung Lam, Russell M. Pitzer, J. Ramanujam, and P. Sadayappan. 2005. Automated Operation Minimization of Tensor Contraction Expressions in Electronic Structure Calculations. In *Computational Science – ICCS 2005 (Lecture Notes in Computer Science)*, Vaidy S. Sunderam, Geert Dick van Albada, Peter M. A. Sloot, and Jack J. Dongarra (Eds.), Vol. 3514. Springer, Berlin, Heidelberg, 155–164. https://doi.org/10.1007/11428831_20
- Frédéric Hecht. 2012. New development in freefem++. *Journal of Numerical Mathematics* 20, 3-4 (2012), 251–266. <https://doi.org/10.1515/jnum-2012-0013>
- Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, Piscataway, NJ, USA, Article 84, 11 pages. <https://doi.org/10.1109/SC.2016.83>

- Bernhard Hientzsch. 2001. *Fast Solvers and Domain Decomposition Preconditioners for Spectral Element Discretizations of Problems in $H(\text{curl})$* . Ph.D. Dissertation. Courant Institute of Mathematical Sciences, New York University. <http://www.cims.nyu.edu/~hientzsc/tr8231.pdf> also Technical Report TR2001-823, December 2001, Department of Computer Science, Courant Institute.
- Michael Holst and Ari Stern. 2012. Geometric Variational Crimes: Hilbert Complexes, Finite Element Exterior Calculus, and Problems on Hypersurfaces. *Foundations of Computational Mathematics* 12, 3 (2012), 263–293. <https://doi.org/10.1007/s10208-012-9119-7> arXiv:1005.4455
- Miklós Homolya and David A. Ham. 2016. A Parallel Edge Orientation Algorithm for Quadrilateral Meshes. *SIAM Journal on Scientific Computing* 38, 5 (2016), S48–S61. <https://doi.org/10.1137/15M1021325> arXiv:1505.03357
- Miklós Homolya, Robert C. Kirby, and David A. Ham. 2017a. Exposing and exploiting structure: optimal code generation for high-order finite element methods. (November 2017). arXiv:1711.02473 Submitted to ACM Transactions on Mathematical Software.
- Miklós Homolya, Lawrence Mitchell, Fabio Luporini, and David A. Ham. 2017b. TSFC: a structure-preserving form compiler. (May 2017). arXiv:1705.03667v1 Submitted to SIAM Journal on Scientific Computing.
- Jeremy Iverson, Chandrika Kamath, and George Karypis. 2015. Evaluation of connected-component labeling algorithms for distributed-memory systems. *Parallel Comput.* 44 (2015), 53–68. <https://doi.org/10.1016/j.parco.2015.02.005>
- A. B. Kahn. 1962. Topological sorting of large networks. *Commun. ACM* 5, 11 (1962), 558–562. <https://doi.org/10.1145/368996.369025>
- George Karniadakis and Spencer Sherwin. 2013. *Spectral/hp element methods for computational fluid dynamics*. Oxford University Press.
- Robert C. Kirby. 2004. Algorithm 839: FIAT, A New Paradigm for Computing Finite Element Basis Functions. *ACM Transactions on Mathematical Software (TOMS)* 30, 4 (2004), 502–516. <https://doi.org/10.1145/1039813.1039820>
- Robert C. Kirby. 2014a. High-Performance Evaluation of Finite Element Variational Forms via Commuting Diagrams and Duality. *ACM Trans. Math. Software* 40, 4, Article 25 (July 2014), 24 pages. <https://doi.org/10.1145/2559983>

- Robert C. Kirby. 2014b. Low-complexity finite element algorithms for the de Rham complex on simplices. *SIAM Journal on Scientific Computing* 36, 2 (2014), A846–A868. <https://doi.org/10.1137/130927693>
- Robert C. Kirby. 2017. A general approach to transforming finite elements. (2017), 28 pages. arXiv:1706.09017
- Robert C. Kirby and Anders Logg. 2006. A compiler for variational forms. *ACM Transactions on Mathematical Software (TOMS)* 32, 3 (2006), 417–444. <https://doi.org/10.1145/1163641.1163644> arXiv:1112.0402
- Robert C. Kirby and Anders Logg. 2007. Efficient Compilation of a Class of Variational Forms. *ACM Trans. Math. Software* 33, 3, Article 17 (Aug. 2007), 20 pages. <https://doi.org/10.1145/1268769.1268771> arXiv:1205.3014
- Robert C. Kirby and Lawrence Mitchell. 2017. Solver composition across the PDE/linear algebra barrier. (2017), 23 pages. arXiv:1706.01346
- Robert C. Kirby and Kieu Tri Thinh. 2012. Fast simplicial quadrature-based finite element operators using Bernstein polynomials. *Numer. Math.* 121, 2 (2012), 261–279. <https://doi.org/10.1007/s00211-011-0431-y>
- Benjamin S. Kirk, John W. Peterson, Roy H. Stogner, and Graham F. Carey. 2006. libMesh: a C++ library for parallel adaptive mesh refinement/coarsening simulations. *Engineering with Computers* 22, 3-4 (2006), 237–254. <https://doi.org/10.1007/s00366-006-0049-3>
- Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA, Article 77 (Oct. 2017), 29 pages. <https://doi.org/10.1145/3133901>
- Andreas Klöckner. 2014. Loo.py: transformation-based code generation for GPUs and CPUs. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY’14)*. ACM, New York, NY, USA, Article 82, 6 pages. <https://doi.org/10.1145/2627373.2627387>
- Andreas Klöckner. 2015. Loo.py: From Fortran to Performance via Transformation and Substitution Rules. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop*

- on Libraries, Languages, and Compilers for Array Programming* (ARRAY 2015). ACM, New York, NY, USA, 1–6. <https://doi.org/10.1145/2774959.2774969>
- Matthew G. Knepley and Dmitry A. Karpeev. 2009. Mesh Algorithms for PDE with Sieve I: Mesh Distribution. *Scientific Programming* 17, 3 (Aug. 2009), 215–230. <https://doi.org/10.1155/2009/948613> arXiv:0908.4427
- Matthew G. Knepley, Karl Rupp, and Andy R. Terrel. 2016. Finite Element Integration with Quadrature on the GPU. (2016). arXiv:1607.04245
- Donald E. Knuth. 1968. *The Art of Computer Programming Vol. 1, Fundamental Algorithms*. Addison Wesley, Reading, MA.
- Martin Kronbichler and Katharina Kormann. 2017. Fast matrix-free evaluation of discontinuous Galerkin finite element operators. (2017). arXiv:1711.03590
- Frank R. Kschischang, Brendan J. Frey, and Hans-Andrea Loeliger. 2001. Factor Graphs and the Sum-Product Algorithm. *IEEE Transactions on Information Theory* 47, 2 (2001), 498–519. <https://doi.org/10.1109/18.910572>
- Chi-Chung Lam, Daniel Cociorva, Gerald Baumgartner, and P. Sadayappan. 1999a. Memory-Optimal Evaluation of Expression Trees Involving Large Objects. In *Proceedings of the 6th International Conference on High Performance Computing (Lecture Notes in Computer Science)*, Prith Banerjee, Viktor K. Prasanna, and Bhabani P. Sinha (Eds.), Vol. 1745. Springer, Berlin, Heidelberg, 103–110. https://doi.org/10.1007/978-3-540-46642-0_15
- Chi-Chung Lam, Daniel Cociorva, Gerald Baumgartner, and P. Sadayappan. 1999b. Optimization of Memory Usage and Communication Requirements for a Class of Loops Implementing Multi-Dimensional Integrals. In *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing (Lecture Notes in Computer Science)*, Larry Carter and Jeanne Ferrante (Eds.), Vol. 1863. Springer, Berlin, Heidelberg, 350–364. https://doi.org/10.1007/3-540-44905-1_22
- Chi-Chung Lam, Thomas Rauber, Gerald Baumgartner, Daniel Cociorva, and P. Sadayappan. 2011. Memory-optimal Evaluation of Expression Trees Involving Large Objects. *Computer Languages, Systems and Structures* 37, 2 (July 2011), 63–75. <https://doi.org/10.1016/j.cl.2010.09.003>

- Chi-Chung Lam, P Sadayappan, and Rephael Wenger. 1996. Optimal Reordering and Mapping of a Class of Nested-Loops for Parallel Execution. In *9th International Workshop on Languages and Compilers for Parallel Computing*. Springer, Berlin, Heidelberg, 315–329. <https://doi.org/10.1007/BFb0017261>
- Chi-Chung Lam, P. Sadayappan, and Rephael Wenger. 1997. On Optimizing a Class of Multi-Dimensional Loops with Reduction for Parallel Execution. *Parallel Processing Letters* 07, 02 (1997), 157–168. <https://doi.org/10.1142/S0129626497000176>
- Michael Lange, Lawrence Mitchell, Matthew G. Knepley, and Gerard J. Gorman. 2016. Efficient Mesh Management in Firedrake Using PETSc DMPlex. *SIAM Journal on Scientific Computing* 38, 5 (2016), S143–S155. <https://doi.org/10.1137/15M1026092> arXiv:1506.07749
- Anders Logg, Kent-Andre Mardal, and Garth Wells (Eds.). 2012a. *Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book*. Lecture Notes in Computational Science and Engineering, Vol. 84. Springer, Berlin, Heidelberg. <https://doi.org/10.1007/978-3-642-23099-8>
- Anders Logg, Kristian B. Ølgaard, Marie E. Rognes, and Garth N. Wells. 2012b. FFC: the FEniCS form compiler. See Logg et al. [2012a], 227–238. <https://doi.org/10.1007/978-3-642-23099-8>
- Kevin Long, Robert C. Kirby, and Bart van Bloemen Waanders. 2010. Unified embedded parallel finite element computations via software-based Fréchet differentiation. *SIAM Journal on Scientific Computing* 32, 6 (2010), 3323–3351. <https://doi.org/10.1137/09076920X>
- Fabio Luporini, David A. Ham, and Paul H. J. Kelly. 2017. An Algorithm for the Optimization of Finite Element Integration Loops. *ACM Trans. Math. Software* 44, 1, Article 3 (March 2017), 26 pages. <https://doi.org/10.1145/3054944> arXiv:1604.05872
- Fabio Luporini, Ana Lucia Varbanescu, Florian Rathgeber, Gheorghe-Teodor Bercea, J. Ramanujam, David A. Ham, and Paul H. J. Kelly. 2015. Cross-Loop Optimization of Arithmetic Intensity for Finite Element Local Assembly. *ACM Transactions on Architecture and Code Optimization* 11, 4 (2015), 57:1–57:25. <https://doi.org/10.1145/2687415> arXiv:1407.0904

- Fredrik Manne and Md. Mostofa Ali Patwary. 2010. A Scalable Parallel Union-find Algorithm for Distributed Memory Computers. In *Proceedings of the 8th International Conference on Parallel Processing and Applied Mathematics: Part I (PPAM'09)*. Springer, Berlin, Heidelberg, 186–195. https://doi.org/10.1007/978-3-642-14390-8_20
- Graham R. Markall, Florian Rathgeber, Lawrence Mitchell, Nicolas Lorient, Carlo Bertolli, David A. Ham, and Paul H. J. Kelly. 2013. Performance-Portable Finite Element Assembly Using PyOP2 and FEniCS. In *Proceedings of 28th International Supercomputing Conference*, Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer (Eds.). Springer, Berlin, Heidelberg, 279–289. https://doi.org/10.1007/978-3-642-38750-0_21
- Dave A. May, Jed Brown, and Laetitia Le Pourhiet. 2014. pTatin3D: High-Performance Methods for Long-Term Lithospheric Dynamics. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 274–284. <https://doi.org/10.1109/SC.2014.28>
- Andrew T. T. McRae, Gheorghe-Teodor Bercea, Lawrence Mitchell, David A. Ham, and Colin J. Cotter. 2016. Automated Generation and Symbolic Manipulation of Tensor Product Finite Elements. *SIAM Journal on Scientific Computing* 38, 5 (2016), S25–S47. <https://doi.org/10.1137/15M1021167> arXiv:1411.2940
- L. S. D. Morley. 1971. The constant-moment plate-bending element. *Journal of Strain Analysis* 6, 1 (1971), 20–24. <https://doi.org/10.1243/03093247V06I020>
- Steffen Müthing, Marian Piatkowski, and Peter Bastian. 2017. High-performance Implementation of Matrix-free High-order Discontinuous Galerkin Methods. (Nov. 2017). arXiv:1711.10885 Submitted to SIAM Journal on Scientific Computing.
- Thomas Nelson, Axel Rivera, Prasanna Balaprakash, Mary Hall, Paul D. Hovland, Elizabeth Jessup, and Boyana Norris. 2015. Generating Efficient Tensor Contractions for GPUs. In *Proceedings of the 2015 44th International Conference on Parallel Processing (ICPP '15)*. IEEE Computer Society, Washington, DC, USA, 969–978. <https://doi.org/10.1109/ICPP.2015.106>
- Kristian B. Ølgaard, Anders Logg, and Garth N. Wells. 2008. Automated Code Generation for Discontinuous Galerkin Methods. *SIAM Journal on Scientific Computing* 31, 2 (2008), 849–864. <https://doi.org/10.1137/070710032>

- Kristian B. Ølgaard and Garth N. Wells. 2010. Optimisations for quadrature representations of finite element tensors through automated code generation. *ACM Trans. Math. Software* 37, 1 (2010), 8:1–8:23. <https://doi.org/10.1145/1644001.1644009> arXiv:1104.0199
- Steven A. Orszag. 1980. Spectral methods for problems in complex geometries. *J. Comput. Phys.* 37, 1 (1980), 70–92. [https://doi.org/10.1016/0021-9991\(80\)90005-4](https://doi.org/10.1016/0021-9991(80)90005-4)
- Anthony T. Patera. 1984. A spectral element method for fluid dynamics: Laminar flow in a channel expansion. *J. Comput. Phys.* 54, 3 (1984), 468 – 488. [https://doi.org/10.1016/0021-9991\(84\)90128-1](https://doi.org/10.1016/0021-9991(84)90128-1)
- Md. Mostofa Ali Patwary, Jean Blair, and Fredrik Manne. 2010. Experiments on Union-find Algorithms for the Disjoint-set Data Structure. In *Proceedings of the 9th International Conference on Experimental Algorithms (SEA'10)*. Springer, Berlin, Heidelberg, 411–423. https://doi.org/10.1007/978-3-642-13193-6_35
- Md. Mostofa Ali Patwary, Peder Refsnes, and Fredrik Manne. 2012. Multi-core Spanning Forest Algorithms Using the Disjoint-set Data Structure. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS '12)*. IEEE Computer Society, Washington, DC, USA, 827–835. <https://doi.org/10.1109/IPDPS.2012.79>
- Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. McRae, Gheorghe-Teodor Bercia, Graham R. Markall, and Paul H. J. Kelly. 2016. Firedrake: Automating the Finite Element Method by Composing Abstractions. *ACM Trans. Math. Software* 43, 3, Article 24 (Dec. 2016), 27 pages. <https://doi.org/10.1145/2998441> arXiv:1501.01809
- Florian Rathgeber, Graham R. Markall, Lawrence Mitchell, Nicolas Lorient, David A. Ham, Carlo Bertolli, and Paul H. J. Kelly. 2012. PyOP2: A High-Level Framework for Performance-Portable Simulations on Unstructured Meshes. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis (SCC '12)*. IEEE Computer Society, Washington, DC, USA, 1116–1123. <https://doi.org/10.1109/SC.Companion.2012.134>
- Pierre-Arnaud Raviart and Jean-Marie Thomas. 1977. A mixed finite element method for 2-nd order elliptic problems. In *Mathematical aspects of finite element methods*,

- Ilio Galligani and Enrico Magenes (Eds.). Lecture Notes in Mathematics, Vol. 606. Springer, 292–315. <https://doi.org/10.1007/BFb0064470>
- Marie E. Rognes, David A. Ham, Colin J. Cotter, and Andrew T. T. McRae. 2013. Automating the solution of PDEs on the sphere and other manifolds in FEniCS 1.2. *Geoscientific Model Development* 6, 6 (2013), 2099–2119. <https://doi.org/10.5194/gmd-6-2099-2013>
- Marie E. Rognes, Robert C. Kirby, and Anders Logg. 2009. Efficient Assembly of $H(\text{div})$ and $H(\text{curl})$ Conforming Finite Elements. *SIAM Journal on Scientific Computing* 31, 6 (2009), 4130–4151. <https://doi.org/10.1137/08073901X> arXiv:1205.3085
- Beverly A. Sanders, Rod Bartlett, Erik Deumens, Victor Lotrich, and Mark Ponton. 2010. A Block-Oriented Language and Runtime System for Tensor Algebra with Very Large Arrays. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. IEEE Computer Society, Washington, DC, USA, 1–11. <https://doi.org/10.1109/SC.2010.3>
- Ravi Sethi and J. D. Ullman. 1970. The Generation of Optimal Code for Arithmetic Expressions. *J. ACM* 17, 4 (1970), 715–728. <https://doi.org/10.1145/321607.321620>
- Edgar Solomonik, Devin Matthews, Jeff Hammond, and James Demmel. 2013. Cyclops Tensor Framework: Reducing Communication and Eliminating Load Imbalance in Massively Parallel Contractions. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS '13)*. IEEE Computer Society, Washington, DC, USA, 813–824. <https://doi.org/10.1109/IPDPS.2013.112>
- Paul Springer and Paolo Bientinesi. 2016. Design of a high-performance GEMM-like Tensor-Tensor Multiplication. (2016). arXiv:1607.00145
- Robert Endre Tarjan. 1975. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM* 22, 2 (April 1975), 215–225. <https://doi.org/10.1145/321879.321884>
- Gavin Truter. 2017. *Structure-preserving automatic differentiation and pull-backs in a language for variational forms*. Master's thesis. Imperial College London.
- Tomáš Vejchodský, Pavel Šolín, and Martin Zítka. 2007. Modular hp-FEM system HERMES and its application to Maxwell's equations. *Mathematics and Computers in*

Simulation 76, 1 (2007), 223–228. <https://doi.org/10.1016/j.matcom.2007.02.001>

Freddie D. Witherden, Antony M. Farrington, and Peter E. Vincent. 2014. PyFR: An open source framework for solving advection–diffusion type problems on streaming architectures using the flux reconstruction approach. *Computer Physics Communications* 185, 11 (2014), 3028–3040. <https://doi.org/10.1016/j.cpc.2014.07.011> arXiv:1312.1638