Imperial College London

Imperial College London

Department of Computing

Abstractions and performance optimisations for finite element methods

Author:
Tianjiao Sun

Supervisors:
Prof Paul H J Kelly
Dr David Ham

Submitted in partial fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London

December 14, 2021

Declaration

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

Tianjiao Sun

Abstract

Finding numerical solutions to partial differential equations (PDEs) is an essential task in the discipline of scientific computing. In designing software tools for this task, one of the ultimate goals is to balance the needs for generality, ease to use and high performance. Domain-specific systems based on code generation techniques, such as Firedrake, attempt to address this problem with a design consisting of a hierarchy of abstractions, where the users can specify the mathematical problems via a high-level, descriptive interface, which is progressively lowered through the intermediate abstractions. Well-designed abstraction layers are essential to enable performing code transformations and optimisations robustly and efficiently, generating high-performance code without user intervention.

This thesis discusses several topics on the design of the abstraction layers of Firedrake, and presents the benefit of its software architecture by providing examples of various optimising code transformations at the appropriate abstraction layers. In particular, we discuss the advantage of describing the local assembly stage of a finite element solver in an intermediate representation based on symbolic tensor algebra. We successfully lift specific loop optimisations, previously implemented by rewriting ASTs of the local assembly kernels, to this higher-level tensor language, improving the compilation speed and optimisation effectiveness.

The global assembly phase involves the application of local assembly kernels on a collection of entities of an unstructured mesh. We redesign the abstraction to express the global assembly loop nests using tools and concepts based on the polyhedral model. This enables us to implement the cross-element vectorisation algorithm that delivers stable vectorisation performance on CPUs automatically. This abstraction also improves the portability of Firedrake, as we demonstrate targeting GPU devices transparently from the same software stack.

Dedication

To Toby, my source of strength and purpose.

Acknowledgements

First and foremost, I would like to thank my supervisors, Prof Paul H J Kelly and Dr David Ham, for their expertise, enthusiasm, vision and generous support throughout my study. I always know that I can rely on their guidance whenever I need it. They lead the research endeavour with such high standards and integrity that I am sure will benefit me for years to come, and for that, I am deeply grateful.

Special thanks to Lawrence Mitchell for his tremendous help in steering my research direction, and for being a reliable source of expert knowledge through the project. His extraordinary productivity is something that I might never attain, but it will always motivate me to strive for doing better.

I am thankful to my fellow developers of the Firedrake project: Miklós Homolya, Andrew McRae and Thomas Gibson, for their help and encouragement in my research journey. A significant part of my research is inspired by the works of Fabio Luporini, who is always there to share with me his ideas and advice. It has been my pleasure to be part of this team.

I had the privilege to interact and work with outstanding researchers in our discipline: J. Ram Ramanujam, Andreas Klöckner, Kaushik Kulkarni. I have benefited tremendously from their thoughts and experiences, and I look forward to continuing our collaboration in the future.

I have been fortunate to share the same office on the first floor of William Penney Laboratory with great colleagues and friends: Emanuele Vespa, Matthew Taylor and Eduardo Carvalho. I will always fondly recall the time we shared, but I am also certain that we will be in touch.

Special thanks to my examiners, Dr Gerard Gorman and Dr Gihan Mudalige, for their kind guidance and encouragement during the Viva.

My funding was provided by the Engineering and Physical Sciences Research Council (EPSRC) through the Centre for Doctoral Training in High Performance Embedded and Distributed Systems (HiPEDS).

Contents

Chapter 1

Introduction

Partial differential equations (PDEs) are essential tools to analyse problems in diverse areas in science and engineering, such as fluid dynamics, electromagnetism, wave propagation and quantum mechanics. Because analytical solutions of PDEs only exit for very few problems of real-world interests, numerical methods, such as finite element methods, finite difference methods and finite volume methods, are critical to applications in research and industry. Compared with other numerical methods, finite element methods provide great flexibility in tailoring the discretisation scheme for specific problems and physical domains. One major downside, however, is the complexity in delivering high-performance implementations on modern computing hardware, which usually requires bespoke hand-tuning, specific to the hardware as well as the application at hand. This challenge in achieving reasonable performance for a wide range of applications limits the adaptation of novel ideas in research in finite element methods.

Domain-specific languages (DSLs) expose a limited set of primitives designed for a particular set of applications to facilitate rapid development and exploration. By restricting the application space, a domain-specific compiler could exploit specialised knowledge in performing transformations that are difficult for general-purpose compilers. In the realm of software systems for solving PDEs using finite element methods, there have been successful examples, such as FEniCS [Logg et al., 2012] and Firedrake [Rathgeber et al., 2015], that employ the design of a multi-layered software stack. The users define the mathematical problems in high-level interfaces, which are lowered through the stack, eventually generating efficient low-level code close to the hardware. At each layer of such a toolchain, a carefully designed intermediate representation is needed to enable specific optimisations and transformations that are more easily realised at the correct levels of abstraction. Compared with a monolithic design, a multi-layered design has the advantage of providing a clean separation of concerns that allows experts in different domains to operate in different levels of abstraction.

1.1 Thesis statement

Well designed abstraction layers for the local and global assembly computations enable effective performance optimisations for finite element solvers.

1.2 Technical contributions

Firedrake is an automated system for the portable solution of PDEs using the finite element method. Firedrake is a publicly available, open-source project that is popular among researchers in numerical mathematics for its versatility and ease to use. This thesis describes the intermediate representations for the local and global assembly computations in Firedrake, and how these representations are exploited and improved to facilitate specific code optimisations. The technical contributions of this thesis are divided into three parts:

- Improvement to local assembly kernel optimisations by raising the abstraction level to perform loop transformations (Chapter 3).

  The local assembly process in finite element methods computes the contribution of the global stiffness matrices and load vectors from a single element in the domain. Such computations can be described in a symbolic tensor language. We show that loop optimisations, previously implemented as manipulation of the Abstract Syntax Trees (ASTs) of the local assembly kernels, can be lifted to tensor computations refactorisation in this language, resulting in a more efficient and robust implementation.

- Effective automated vectorisation of the global assembly of matrix-free operators (Chapter 4).

  The global assembly process computes the global stiffness matrices and load vectors by aggregating the contributions from all the elements in the domain. By introducing a polyhedral-like abstraction of loop nests to represent the global assembly computations, we successfully implemented the cross-element vectorisation algorithm that delivers robust, portable vectorisation performances on modern CPUs.

- GPU code generation for the global assembly of matrix-free operators (Chapter 5).

  We show that the abstraction of the loop nests, mentioned above, facilitates targeting GPU devices in Firedrake by generating CUDA and OpenCL kernels from the same high-level description of the global assembly computations.

1.3 Dissemination

The work described in this thesis has been made available in open-source software packages, and disseminated in the following publications and conference presentations:

- *Loop fusion for finite element assembly in PyOP2.* FEniCS Workshop. Luxembourg, 2017.
- *Automated cross-element vectorization in Firedrake.* FEniCS Workshop. Oxford, UK, 2018.
- *Automated cross-element vectorization in Firedrake.* Firedrake Workshop. London, UK, 2018.
- *Automated cross-element vectorization in Firedrake.* Dagstuhl Seminar on Loop Optimization. Schloss Dagstuhl, Germany, 2018.
- *Automated cross-element vectorization in Firedrake.* Workshop on Compilers for Parallel Computing, Dublin, Ireland, 2018.
- *Firedrake: Automated High Performance Finite Element Simulation.* SIAM Conference on Computational Science and Engineering. Spokane, USA, 2019.
- *Performance Optimisation of Finite Element Assembly in Firedrake.* SIAM Conference on Computational Science and Engineering. Spokane, USA, 2019.
- *SIMD vectorization for matrix-free finite element methods.* Intel Extreme Performance Users Group. Web seminar, 2019.
- [Sun, Mitchell, Kulkarni, Klöckner, Ham, and Kelly, 2020]. *A study of vectorization for matrix-free finite element methods.* International Journal of High Performance Computing Applications 34(6): 629-644.

1.4 Thesis outline

This thesis describes investigations into improving the abstractions of the software stack of Firedrake. Chapter 2 introduces the finite element method and the Firedrake software stack, with the emphasis on how the components are organised into different abstraction layers. Chapter 3 describes our approach to perform loop optimisations on local assembly kernels in the higher abstraction layer based on tensor algebra. Chapter 4 describes the cross-element vectorisation algorithm, which achieves reliable and effective vectorisation for the global assembly of matrix-free operators on CPUs. This algorithm is enabled by integrating with Loopy [Klöckner, 2014] in Firedrake as a new abstraction layer to represent the global assembly computation. We show that this representation provides a pathway to target GPU devices in Firedrake in Chapter 5. Each of the above chapters includes a section on conducting experimental evaluations on our approaches. Finally, Chapter 6 concludes this thesis and discusses possible future research directions.

Chapter 2

Background and related work

In this chapter, we provide the relevant background materials to the topics covered in subsequent chapters. These include the background on the finite element method, a review of relevant software tools, and a summary of the software architecture of Firedrak.

2.1 Finite element methods

The finite element method (FEM) is one of the most extensively used numerical methods to solve real-world science and engineering problems that are governed by partial differential equations. This popularity is primarily due to its ability to handle complex geometries and generalisability to a wide range of problems. In this section, we review key mathematical and computational concepts of finite element methods that are relevant to the rest of this thesis.

2.1.1 Variational formulation

We follow the notation introduced by Logg et al. [2012] and Luporini et al. [2017]. Consider the symmetric positive-definite Helmholtz equation of some unknown field $u : \Omega \to \mathbb{R}$ defined on a domain $\Omega \subset \mathbb{R}^d$ with boundary $\partial\Omega$:

$$\nabla^2 u + u = f \quad \text{in } \Omega, \tag{2.1}$$

where f is a given function defined on Ω. We prescribe a Dirichlet boundary condition on $\Gamma_D \subseteq \partial\Omega$ and a Neumann boundary condition on $\Gamma_N \subseteq \partial\Omega$, such that $\Gamma_D + \Gamma_N = \partial\Omega$:

$$\begin{aligned} u &= u_0 \qquad \text{on } \Gamma_D, \\ \partial_n u &= g \qquad \text{on } \Gamma_N, \end{aligned} \tag{2.2}$$

where u_0 is a given function and $\partial_n u$ denotes the partial derivative of u in the normal direction.

To obtain the *variational* or *weak* formulation of Equation (2.1), we multiply (2.1) by a suitable test function v:

$$\int_\Omega v\nabla^2 u + uv \,\mathrm{d}x = \int_\Omega fv \,\mathrm{d}x. \tag{2.3}$$

If we let the test function v vanish on the Dirichlet boundary Γ_D, after integrating the left-hand side by parts and applying the divergence theorem, we obtain

$$\int_\Omega \nabla u \cdot \nabla v + uv \,\mathrm{d}x - \int_{\Gamma_N} gv \,\mathrm{d}s = \int_\Omega fv \,\mathrm{d}x. \tag{2.4}$$

Note that for Equation (2.4) to be well-defined, we only require the first derivatives (defined weakly) of u and v to be square-integrable on Ω, that is to say, u and v must be functions from the Sobolev space $H^1(\Omega)$.

After moving the unknown terms to the left-hand side and other terms to the right-hand side, we obtain the *variational* or *weak* formulation of Equation (2.1):

$$\begin{gathered}\text{Find } u \in V \text{ such that}\\ \int_\Omega \nabla u \cdot \nabla v + uv \,\mathrm{d}x = \int_\Omega fv \,\mathrm{d}x + \int_{\Gamma_N} gv \,\mathrm{d}s \quad \forall v \in \hat{V},\end{gathered} \tag{2.5}$$

where $\hat{V}$, the test function space, is defined as

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

and V, the trial function space, is defined by shifting $\hat{V}$ by the Dirichlet boundary condition

$$V = \{v \in H^1(\Omega) : v = u_0 \text{ on } \Gamma_D\}. \tag{2.7}$$

It is worth noting that the different types of boundary conditions are treated in different ways: the Neumann boundary conditions become part of the variational formulation, whereas the Dirichlet boundary conditions are imposed by restricting the trial function space.

More abstractly, a variational formulation of a partial differential equation can be expressed as:

$$\begin{gathered}\text{Find } u \in V \text{ such that}\\ a(u, v) = L(v) \quad \forall v \in \hat{V},\end{gathered} \tag{2.8}$$

where $a(\cdot,\cdot)$ is a bilinear form (i.e. a bilinear map $V \times \hat{V} \to \mathbb{R}$), $L(\cdot)$ is a linear form (i.e. a linear map $\hat{V} \to \mathbb{R}$), V and $\hat{V}$ are the suitably chosen function spaces of the trial and test functions.

2.1.2 Finite element discretisation

When using finite element methods to solve partial differential equations numerically, we discretise the function spaces to find an approximate solution. This approach contrasts with the finite difference methods, which discretises the differential operators in the equations.

Let V_h be a finite-dimensional subspace of V with dimension N. Assuming $V = \hat{V}$, using Galerkin projection (see, for example, [Brenner and Scott, 2007]), we can find the approximate solution u_h of Equation (2.8) by solving the following projection equation:

$$\begin{gathered}\text{Find } u_h \in V_h \text{ such that}\\ a(u_h, v_h) = L(v_h) \quad \forall v_h \in V_h.\end{gathered} \tag{2.9}$$

To estimate the error of this approximate solution, we recall the Galerkin orthogonality theorem, which states that the error function, defined as $\epsilon_h = u - u_h$, is always orthogonal to the approximate function space V_h:

$$a(\epsilon_h, w) = 0 \quad \forall w \in V_h. \tag{2.10}$$

Furthermore, if we define the *energy norm* of a space V as

$$\|v\|_E = \sqrt{a(v,v)} \quad \forall v \in V, \tag{2.11}$$

then the energy norm is guaranteed to be minimised by u_h:

$$\|u - u_h\|_E = \min\{\|u - v\|_E : v \in V_h\}. \tag{2.12}$$

That is to say, u_h is the best approximation in the subspace V_h to the exact solution u of the original equation, with the error measured according to the energy norm.

Let $\{\phi_i\}_{i=1}^N$ be the set of basis functions spanning V_h. Then $u_h \in V_h$ can be expressed as a linear combination of ϕ_i:

$$u_h = \sum_{i=1}^{N} u_i \phi_i. \tag{2.13}$$

For a fixed set of basis functions, we can use the vector of the basis coefficients $\mathbf{u} = (u_1, \cdots, u_n)$ as a equivalent representation of u for simplicity. Substituting each basis function ϕ_i as the test function v in Equation (2.9), we can rewrite (2.9) as the following linear system

$$\mathbf{A}\mathbf{u} = \mathbf{b}, \tag{2.14}$$

where the *stiffness matrix* $\mathbf{A}$ and *load vector* $\mathbf{b}$ are the linear algebra representations of the bilinear operator a and linear operator L. The elements of $\mathbf{A}$ and $\mathbf{b}$ are computed as:

$$\begin{aligned}\mathbf{A}_{ij} &= a(\phi_i, \phi_j), & i, j &= 1 \dots N,\\ \mathbf{b}_i &= L(\phi_i), & i &= 1 \dots N.\end{aligned} \tag{2.15}$$

In Firedrake, the users define the equations and discretisations in Unified Form Language (UFL) [Alnæs et al., 2014], a domain-specific language embedded in Python. Listing 2.1 shows an excerpt of UFL code to define and solve the variational form of the Helmholtz equation (2.4) in Firedrake. Here the domain is a 10×10 triangulation of a unit square. We choose the first-order Lagrange element as our approximation space (i.e. the space of linear polynomials). UFL is a symbolic language that allows the users to define the problems in a high-level, intuitive interface close to mathematics. Firedrake, as explained in the following sections, computes the stiffness matrix and load vector corresponding to the discretised equation, and leverages the solvers provided by the PETSc library [Balay et al., 2017] to solve the resulting linear systems.

Listing 2.1: Solving the variational form of the Helmholtz equation in Firedrake on a unit square. The definition of `f`, the given function in Equation (2.1) is omitted for simplicity. The numerical solution is assigned to `u`.

```python
mesh = UnitSquareMesh(10, 10)
V = FunctionSpace(mesh, "Lagrange", 1)
u = TrialFunction(V)
v = TestFunction(V)
a = (dot(grad(v), grad(u)) + v*u) * dx
L = f * v * dx
u = Function(V)
solve(a, L, u)
```

2.1.3 Local and global assembly

The computation of matrix $\mathbf{A}$ and vector $\mathbf{b}$ according to Equation (2.15) is referred to as the *assembly* of $\mathbf{A}$ and $\mathbf{b}$. Let $\mathcal{T}$ be a tessellation of the domain Ω, and N be the dimension of the test and trial function space (N is usually at least 10^6 in real applications), a naive way to compute $\mathbf{A}$ according to (2.15) is shown in Algorithm 1.

Algorithm 1 Naive assembly algorithm of a bilinear form a according to its definition using the basis functions.

```
for $i \leftarrow 1, \ldots, N$ do
    for $j = 1, \ldots, N$ do
        $\mathbf{A}_{ij} \leftarrow a(\phi_i, \phi_j)$
    end for
end for
```

However, this algorithm does not take advantage of the sparsity structure of matrix $\mathbf{A}$. Because the basis functions in the chosen function space of finite element methods only have local support, we know that $a(\phi_i, \phi_j) = 0$ for a pair of basis functions ϕ_i and ϕ_j, unless they have support in the same element $K \in \mathcal{T}$.

Therefore, a more efficient algorithm to assemble $\mathbf{A}$ is by iterating over all the elements in the domain and accumulate the contribution from each element, skipping all pairs of basis functions that do not have support on the same element:

$$\int_\Omega a(\phi_i, \phi_j)\,\mathrm{d}x = \sum_{K \in \mathcal{T}} \int_K a(\phi_i, \phi_j)\,\mathrm{d}x. \tag{2.16}$$

Let $\{\phi_i^K\}_{i=1}^{N_l}$ be the subset of $\{\phi_i\}_{i=1}^{N}$ that are supported on the element K, we can define the *dense* matrix $\mathbf{A}^K$ associated with element K as

$$\mathbf{A}_{ij}^K = a_K(\phi_i^K, \phi_j^K), \tag{2.17}$$

where

$$a_K(u, v) = \int_K \nabla u \cdot \nabla v + uv\,\mathrm{d}x \tag{2.18}$$

is the original bilinear form a restricted to K.

This approach, adapted from [Logg et al., 2012], is described in Algorithm 2. The number function $m(K, i)$ enables identifying a local basis function ϕ_i^K as the local restriction of the global basis function $\phi_{m(K,i)}$. This is achieved by mapping the local numbering i of the basis function ϕ_i^K to the global numbering $m(K, i)$ of the same basis function.

Algorithm 2 Global assembly algorithm by iterating over all elements in the domain and accumulating local contributions, taking advantage of the property that the basis functions have local support.

```
$A \leftarrow 0$
for all $K \in \mathcal{T}$ do
    Compute $A^K$
    for $i = 1, \dots, N_l$ do
        for $j = 1, \dots, N_l$ do
            $\mathbf{A}_{m(K,i),m(K,j)} \leftarrow \mathbf{A}_{m(K,i),m(K,j)} + A_{ij}^K$
        end for
    end for
end for
```

We can identify two parts of the overall assembly process of matrix $\mathbf{A}$: we name the computation of local matrix $\mathbf{A}^K$ according to Equation (2.17) *local* assembly, whereas the process of accumulating local contributions to the global matrix $\mathbf{A}$ is named *global* assembly.

Local assembly as tensor contraction

In Firedrake, the local assembly process is accomplished by the Two-Stage Form Compiler (TSFC) [Homolya et al., 2018]. Abstractly, TSFC takes a differential form (either a bilinear form or a linear form) as the input and generates the code that computes a local tensor representing the contribution of the global stiffness matrix or load vector from a given element in the domain. This generated routine is often called a *local assembly kernel*. Below, we highlight some of the essential computational aspects of local assembly kernels.

Numerical integration with quadrature rules As shown in Equation (2.18), a local assembly kernel computes entries of the local tensor by evaluating integrals. Because symbolic integration is only feasible in very few cases, such integrals often need to be evaluated numerically. The general method that TSFC generates to compute integrals is by applying numerical quadratures.

A numerical quadrature rule comprises of a set of N_q quadrature points at coordinates $\{X_q\}_{q=1}^{N_q}$ and a corresponding set of quadrature weights $\{w_q\}_{q=1}^{N_q}$. To approximately evaluate the integral of a function f over domain D, we can write the integral as the sum of the function values at the quadrature points, weighted by the quadrature weights:

$$\int_D f \,\mathrm{d}x \approx \sum_{q=1}^{N_q} f(X_q) w_q. \tag{2.19}$$

If f is a polynomial, the integral can be computed precisely (up to machine precision) if N_q is large enough. TSFC chooses the quadrature rule to evaluate the integral based on the expression to be integrated and discretisation function space. Users can overwrite this decision through runtime parameters passed to TSFC.

Reference space and pull-back In order to construct a global function space, i.e. one that is defined on the whole domain Ω, it is common to build a single *reference* finite element function space, defined on a reference element $\hat{K}$, and maps it to every element in Ω through coordinate transformations.

For each element $K \in \Omega$, let the mapping $F_K : \hat{K} \to K$ be the transformation from $\{\hat{x}\}$, the coordinate system of the reference element $\hat{K}$, to $\{x\}$, the coordinate system of the element K. For simplicial shapes (e.g. triangles and tetrahedra), this mapping is an affine transformation.

The *pull-back* associated with the transformation F_K, F^* is defined as

$$F^*(v)(\hat{x}) = v(F_K(\hat{x})) \quad \forall v \in V_K, \tag{2.20}$$

where V_K is the global function space restricted to K. In other words, the pull-back maps a function in the global function space to a function in the reference function space:

$$F^*(v) := v \circ F_K. \tag{2.21}$$

We continue to follow the example of assembling the bilinear form of the Helmholtz operator. Let $\{\Phi_i\}_{i=N_l}^{N_l}$ be the set of basis functions in the reference space. The basis functions of the global function space, $\{\phi_i^K\}_{i=N_l}^{N_l}$, can be transformed to the reference space by applying the reverse of the pull-back:

$$\phi_i^K = \Phi_i \circ F_K^{-1}. \tag{2.22}$$

We can also obtain the spatial gradients of the basis functions by applying the chain rule to (2.22):

$$\nabla \phi_i^K = J_K^T \hat{\nabla} \Phi_i \circ F_K^{-1}, \tag{2.23}$$

where $J_K = \hat{\nabla} F_K$ is the Jacobian of the transformation of the coordinate system. The operator $\hat{\nabla}$ indicates differentiation with respect to the coordinates of the reference element, $\{\hat{x}\}$.

The measure of the integral also needs to be scaled by $|\det(J_k)|$. Finally, we can rewrite the integral in an arbitrary element K in Equation (2.17) as an integral in the reference element $\hat{K}$:

$$\mathbf{A}_{ij}^K = \int_{\hat{K}} \left(J_K^T \nabla \hat{\Phi}_i \cdot J_K^T \hat{\nabla} \Phi_j + \Phi_i \Phi_j \right) |\det(J_K)| \, \mathrm{d}\hat{x}. \tag{2.24}$$

Using a suitable quadrature rule $\{X_q,\ w_q\}_{q=1}^{N_q}$, the above integral can be evaluated numerically as the following summation:

$$\mathbf{A}_{ij}^K = \sum_{q=1}^{N_q} w_q \left(J_K^{-T}(X_q) \hat{\nabla} \Phi_i(X_q) \cdot J_K^{-T}(X_q) \hat{\nabla} \Phi_j(X_q) + \Phi_i(X_q) \Phi_j(X_q) \right) |\det(J_K)| \tag{2.25}$$

Tensor contraction and GEM In Equation (2.25), terms such as $\Phi_i(X_q)$ and $\hat{\nabla}\Phi_i(X_q)$ are the evaluation of the reference basis functions and their gradients at quadrature points in the reference element. These expressions do not depend on the cell K and can be pre-computed separately. TSFC relies on FIAT [Kirby, 2004] and FInAT [Homolya et al., 2017] to compute these tensors, called *tabulations*, for a given reference function space and quadrature rule.

FIAT supports a wide range of finite element discretisations of arbitrary polynomial degrees. Given the coordinates of some points in the reference element, FIAT computes the evaluations of the basis functions and their gradients at these points. FInAT performs similar functions but also preserves the internal structure of the finite elements (e.g. for tensor product elements, which are constructed by composing discretisations of lower physical dimensions, the tabulation tensors can be written as tensor products of tensors of lower ranks). These structures can be exploited for optimisations in code generation.

To simplify the notation and to highlight the free indices in Equation (2.25), we write these tabulation tensors as

$$\begin{aligned}\mathbf{\Phi}_{iq} &= \Phi_i(X_q), \\ \nabla\mathbf{\Phi}_{ikq} &= \frac{\partial \Phi_i}{\partial \hat{x_k}}(X_q).\end{aligned} \tag{2.26}$$

The Jacobian of the coordinate transformation at each quadrature point can be written as the tensor

$$\mathbf{J_K}_{ikq} = \frac{\partial x_i}{\partial \hat{x_k}}(X_q), \quad i,k = 1\dots d, \tag{2.27}$$

and the inverse of the Jacobian can be written as the tensor

$$\mathbf{J_K^{-1}}_{ikq} = \frac{\partial \hat{x}_i}{\partial x_k}(X_q) \quad , i,k = 1\dots d, \tag{2.28}$$

with d the spatial dimension.

Rearranging the summations and expanding the dot product, we can write Equation (2.25) as the following scalar expression that computes each entry of the matrix $\mathbf{A}^K$:

$$\mathbf{A}_{ij}^K = \sum_{q=1}^{N_q}\sum_{\alpha=1}^{d}\sum_{\beta=1}^{d}\sum_{\gamma=1}^{d} w_q \left(\mathbf{J_K^{-1}}_{\alpha\beta q}\nabla\mathbf{\Phi}_{i\beta q}\mathbf{J_K^{-1}}_{\alpha\gamma q}\nabla\mathbf{\Phi}_{j\gamma q} + \mathbf{\Phi}_{iq}\mathbf{\Phi}_{jq}\right)\left|\det(\mathbf{J_K}_q)\right|. \tag{2.29}$$

With the exception of computing the expression $\left|\det(\mathbf{J_K}_q)\right|$, Equation (2.29) is an ordinary tensor contraction. TSFC incorporates GEM, an intermediate representation for tensor algebra, to express local assembly computations as tensor contractions. In GEM, expressions are represented as directed acyclic graphs (DAGs) on tensor nodes. We discuss the optimisations of the representations in GEM in more depth in Chapter 3.

Except for the Jacobian tensors $\mathbf{J_K}$ and $\mathbf{J_K^{-1}}$, which need to be computed from $\mathbf{x^K}$, the coordinates of each element K in the domain, all other tensors in (2.29) are compile-time constants. We note that there are many possible orders to evaluate such tensor contraction expressions as multi-dimensional loops. One possible approach is listed in Algorithm 3. Note that in the cases of simplicial geometry, since the coordinate transformation is constant within each cell, $\mathbf{J_K}$ and $\mathbf{J_L^{-1}}$ are independent of the quadrature number q, and can be hoisted out of the q loop.

Arguments and coefficients In general, a multi-linear differential form could also depend on functions other than the basis functions of the test and

Algorithm 3 Local assembly algorithm for the Helmholtz operator

```
function HELMHOLTZ($\mathbf{x}^{\mathbf{K}}$)
    Define $\mathbf{\Phi}, \nabla\mathbf{\Phi}, w$                                ▷ Constant tensors
    $A^K \leftarrow 0$
    for $q = 1, \ldots, N_q$ do
        Compute $\mathbf{J}_{\mathbf{K}}$ from $\mathbf{x}^{\mathbf{K}}$              }
        Compute $\mathbf{J}_{\mathbf{K}}^{-1}$ from $\mathbf{J}_{\mathbf{K}}$          }  Independent of $q$ on
        Compute $\left|\det(\mathbf{J}_{\mathbf{K}})\right|$ from $\mathbf{J}_{\mathbf{K}}$     }  simplicial geometry
        for $i, j = 1, \ldots, N_l$ do
            for $\alpha, \beta, \gamma = 1, \ldots, d$ do
                $A^K_{ij} \leftarrow A^K_{ij} + \left(w_q {\mathbf{J}_{\mathbf{K}}^{-1}}_{\alpha\beta q} \nabla\mathbf{\Phi}_{i\beta q} {\mathbf{J}_{\mathbf{K}}^{-1}}_{\alpha\gamma q} \nabla\mathbf{\Phi}_{j\gamma q} + \mathbf{\Phi}_{iq}\mathbf{\Phi}_{jq}\right) \left|\det(\mathbf{J}_{\mathbf{K}})\right|$
            end for
        end for
    end for
    return $A^K$
end function
```

trial function spaces. These functions are potentially members of function spaces other than the test and trial function spaces.

In the UFL terminology, the test and trial (basis) functions are called *arguments*, whereas other arbitrary functions in a multi-linear differential form are called *coefficients*. A multi-linear form is always linear with respect to its arguments, but can be non-linear with respect to its coefficients. Consider the weighted Laplacian operator $\mathcal{L}$ in a heat transform equation, defined as

$$\mathcal{L}(u) = -\nabla \cdot (\kappa \nabla u), \tag{2.30}$$

where $\kappa : \Omega \to \mathbb{R}$ is a function that usually represent the thermal conductivity of the object. The bilinear form of the operator $\mathcal{L}$ is

$$a(u, v) = \int_\Omega \kappa \nabla u \cdot \nabla v \, \mathrm{d}x, \tag{2.31}$$

where u is the trial function and v is the test function. In the above equation, u and v are the arguments of $a(\cdot, \cdot)$ and κ is a coefficient.

In order to assemble a multi-linear form with coefficients, TSFC requires the evaluation of the coefficient functions at all the quadrature points. Assuming κ is a member of the function space U with basis functions $\{\psi_i\}_{i=1}^N$, the restriction of κ on an element K, κ^K, can be written as the weighted sum of the basis functions:

$$\kappa^K(x) = \sum_i^{N_l} {\kappa^{\mathbf{K}}}_i \psi_i(x), \tag{2.32}$$

where $\kappa^{\mathbf{K}}$ is the vector of the weights of the basis functions.

We follow the same method applied to the arguments to evaluate κ^K in the reference element. Let $\{\Psi_i\}_{i=1}^{N_l}$ be basis functions of the reference function space, and $\hat{\kappa}$ be the pull-back of κ^K in the reference element, the value of $\hat{\kappa}$ at a quadrature point q is

$$\hat{\kappa}_q = \sum_{i=1}^{N_l} {\kappa^{\mathbf{K}}}_i \mathbf{\Psi}_{iq}, \tag{2.33}$$

with the tabulation tensor $\mathbf{\Psi}$ defined similarly to Equation (2.26), but using $\{\Psi_i\}_{i=1}^{N_l}$ instead of $\{\Phi_i\}_{i=1}^{N_l}$ as the basis functions.

To assemble forms with coefficients using numerical quadratures, we require the evaluations of the coefficients at the quadrature points. This can be computed from the input tensor $\kappa^{\mathbf{K}}$ as shown in (2.33).

The local assembly of the weighted Laplacian operator in Equation (2.31) is computed as the following tensor contraction:

$$\mathbf{A}_{ij}^K = \sum_{q=1}^{N_q}\sum_{i=1}^{N_l} {\kappa^{\mathbf{K}}}_i \mathbf{\Psi}_{iq} \sum_{\alpha=1}^{d}\sum_{\beta=1}^{d}\sum_{\gamma=1}^{d} w_q {\mathbf{J}_{\mathbf{K}}^{-1}}_{\alpha\beta q} \nabla\mathbf{\Phi}_{i\beta} {\mathbf{J}_{\mathbf{K}}^{-1}}_{\alpha\gamma q} \nabla\mathbf{\Phi}_{j\gamma} \left|\det(\mathbf{J}_{\mathbf{K}q})\right| . \tag{2.34}$$

The tabulation tensor $\mathbf{\Psi}_{iq}$ can be obtained from FIAT and FInAT at compile time, whereas the tensor ${\kappa^{\mathbf{K}}}_i$ needs to be passed in as additional parameters to the local assembly routine.

In summary, the local assembly of differential forms with coefficients requires additional inputs to the kernel and introduces more reduction indices in the tensor contraction computation.

Global assembly as parallel loops over unstructured meshes

During the global assembly phase, the local contribution from each mesh entity, computed by the local assembly kernel generated by TSFC, is accumulated into the global data structures. This is illustrated in Algorithm 2. In Firedrake, the global assembly stage is handled by PyOP2 [Rathgeber et al., 2012]. The abstraction of PyOP2 is the programming model of parallel computations over entities on an unstructured mesh. Mesh entities are modelled as sets of integers, and the connections of the entities are modelled as maps between integer sets, representing the mesh topology. During global assembly, PyOP2 is responsible for iterating over the mesh entities, marshalling data in and out of the local assembly kernels and handling synchronisations among parallel compute nodes.

PyOP2 programs are organised as parallel loops, or `parloops`. A `parloop` specifies a computational kernel, a set of mesh entities over which the kernel is applied, and all the input and output data of the kernel. The data are associated with the mesh entities: each piece of data can be directly defined on a mesh entity, or indirectly accessed through a mapping defined on mesh entities.

Date objects also carry access access descriptors, such as `WRITE`, `READ`, `INC`, which indicates their access pattern in the kernel, to facilitate scheduling and lazy evaluation of the `parloops`.

As an example, the `parloop` for the global assembly of the bilinear form of the Helmholtz operator, discretised using first order Lagrange finite element on a triangular mesh is:

```
Parloop(helmholtz, cells, A(cell2vert, INC), coords(cell2vert, READ))
```

`helmholtz` is the local assembly kernel in Algorithm 3, generated by TSFC. `cells` is the set of all triangles in the mesh. `A` is the global (sparse) matrix that holds the global assembly result. `coords` is the global data structure that holds the coordinates of the vertices of the triangles. These are needed in computing the Jacobian of the coordinate transformation from a triangle in the mesh to the reference element. The mapping `cell2vert` maps each triangle to its vertices and provides the indirect access from a triangle to the coordinates of its vertices, stored in `coords`. Because the basis functions of first-order Lagrange finite element on triangles are defined only on the vertices, `cell2vert` is also used to indirectly access `A` in this `parloop`.

When performing the computation, PyOP2 iterates over the set `cells`, gathers the input data using the access mappings into local (dense) arrays, invokes the computation kernel with the input data, and finally, updates the (sparse) global data structure using the access mappings with the results produced by the kernel. In our example, these local arrays correspond to the tensors A^K and $\mathbf{x^K}$ respectively in Algorithm 3.

2.1.4 The matrix-free methods

Computationally, the assembly of stiffness matrices becomes more expensive for higher-order methods, in terms of both space and time. One observation is that the stiffness matrices are only needed to solve the linear systems representing the differential forms, and are not otherwise required. The matrix-free method leverages the property that when solving linear systems using Krylov methods, it is sufficient to be able to compute the result of multiplying the stiffness matrix with an arbitrary vector. Since the entries to the stiffness matrices themselves are usually not required[1], the explicit assembly of the matrices can be avoided.

Compared with assembling the stiffness matrices, matrix-free methods perform more computation per matrix-vector product, but benefit from much lower startup cost and smaller memory footprint. These methods

[1]Some preconditioners such as algebraic multigrid, however, do require the entries of the stiffness matrices.

are usually more suitable for modern hardware because improvement in the memory bandwidth has generally lagged that of the advancement in computing power in high-performance computing systems.

Consider a bilinear form $a(\cdot,\cdot) : V \times V \to \mathbb{R}$ and the sparse matrix $\mathbf{A}$ that represents the action of $a(\cdot,\cdot)$ in a discretised function space U. An arbitrary function $x \in U$ can be written as a weighted sum of the basis functions $\{\phi_i\}_{i=1}^{N}$ of U,

$$x = \sum_{i=1}^{N} x_i \phi_i. \tag{2.35}$$

The vector $\mathbf{x} = (x_1, \cdots, x_N)$ is therefore the linear algebra representation of the function x. To solve the linear system $\mathbf{Au} = \mathbf{b}$ with a matrix-free method, we need to provide the solver with a routine that computes the matrix-vector product $\mathbf{Ax}$, given an arbitrary vector $\mathbf{x}$. This product is also called the *action* of the operator a on the function x.

According to the definition of $\mathbf{A}$ in (2.15), and making use of the linearity of the operator a, $\mathbf{Ax}$ can be computed as

$$\begin{aligned}(\mathbf{Ax})_i &= \sum_{j=1}^{N} \mathbf{A}_{ij} x_j = \sum_{j=1}^{N} a(\phi_i, \phi_j) x_j \\ &= a(\phi_i, \sum_{j=1}^{N} x_j \phi_j) = a(\phi_i, x).\end{aligned} \tag{2.36}$$

Abstractly, the assembly of the action of a on x is the same as the assembly of the one form $a_x \equiv a(\cdot, x) : V \to \mathbb{R}$. The function $\mathbf{x}$ is simply a coefficient to the local assembly kernel.

In Firedrake, the users can switch to matrix-free approaches by configuring the solver options, without having to change the high-level problem definitions. This is described in detail by Kirby and Mitchell [2018].

Putting all the concepts together, we arrive at the global assembly routine in Algorithm 4. We assume that the Jacobian is constant on each element in the domain. Einstein notation is used in the tensor contraction expressions for simplicity. The set of contracted indices is $\{\alpha, \beta, \gamma, q, j\}$, and the set of free indices is $\{i\}$. The algorithmic complexity of this routine is therefore

$$N_K \times N_b^2 \times N_q \times d^3, \tag{2.37}$$

where N_K is the number of elements in the domain, N_b is the number of basis functions in the reference function space, N_q is the number of quadrature points, and d is the dimension of the physical space.

Algorithm 4 Global assembly routine for the (matrix-free) action of the Helmholtz operator, on a simplicial element. Einstein notation is used for tensor contractions. $\mathcal{T}$ is the set of all elements in the domain. $\mathbf{x}, \mathbf{f}, a$ are the global arrays that hold the coordinates of the cell vertices, the input function (expressed as weights of basis functions) and the output function (expressed weights of basis functions). $m_\mathbf{x}, m_\mathbf{f}, m_\mathbf{a}$ are the mappings from elements to their associated data entries in the global arrays.

```
function HELMHOLTZ($\mathcal{T}, \mathbf{x}, m_\mathbf{x}, \mathbf{f}, m_\mathbf{f}, a, m_a$)
    Define $\mathbf{\Phi}, \nabla\mathbf{\Phi}, \mathbf{\Psi}, w$                ▷ Constant tensors
    for all $K \in \mathcal{T}$ do
        $\mathbf{x^K} \leftarrow \mathbf{x}_{m_\mathbf{x}(\mathbf{K})}$        } Gather data for cell $K$
        $\mathbf{f^K} \leftarrow \mathbf{f}_{m_\mathbf{f}(\mathbf{K})}$
        Compute $\mathbf{J_K}$ from $\mathbf{x^K}$             }
        Compute $\mathbf{J_K^{-1}}$ from $\mathbf{J_K}$          } Jacobian computation
        Compute $|\det(\mathbf{J_K})|$ from $\mathbf{J_K}$     }
        $a_i^K \leftarrow w_q \left(\mathbf{f^K}_j \Psi_{jq}\right) \left(\mathbf{J_K^{-1}}_{\alpha\beta q} \nabla\mathbf{\Phi}_{i\beta q} \mathbf{J_K^{-1}}_{\alpha\gamma q} \nabla\mathbf{\Phi}_{j\gamma q} + \mathbf{\Phi}_{iq}\mathbf{\Phi}_{jq}\right) |\det(\mathbf{J_K})|$
        $a_{m_a(\mathbf{K})} \leftarrow a_{m_a(\mathbf{K})} + a^K$
    end for
end function
```

2.2 Automating PDE solvers

The development of high-performance PDE solvers is a challenging undertaking, and over the years, many programming models and libraries have been introduced to aid the programmers and improve performance portability in the landscape of ever-changing CPUs and accelerators. These include pragma based language extensions such as OpenMP [OpenMP Architecture Review Board, 2018] and OpenACC [OpenACC Organization], as well as embedded languages, such as SYCL [Khronos Group]. Library based solutions such as Kokkos [Edwards et al., 2014], RAJA [Beckingsale et al., 2019] and Thrust [Hoberock and Bell, 2010] offer higher level interfaces using generic programming.

Specialised tools have been developed in the domain of finding numerical solutions to PDEs. Due to the simplicity of finite difference methods (FDM), many projects using FDM adopt automated code generation to generate optimised stencil kernels from a high level description of the computation. Examples in this area include ExaStencils [Lengauer et al., 2014], OpenSBLI [Jacobs et al., 2017] and Devito [Luporini et al., 2018]. Many of these projects are domain-specific languages (DSLs) that are embedded in a host language, with Python one of the popular choices. Beyond FDM projects, other examples of DSLs and frameworks include OpenFOAM [The

OpenFOAM Foundation, 2018], which uses finite volume methods, PyFR [Witherden et al., 2014], which uses flux reconstruction methods, and Dune [Dedner et al., 2010] and Deal.II [Bangerth et al., 2007], which are object-oriented C++ finite element packages.

In the realm of frameworks that support the finite element method, OP2 [Mudalige et al., 2012] facilitates the global assembly process with an abstraction for representing computations on an unstructured mesh. Since FEM fundamentally relies on the theory of function spaces, some projects using FEM introduce high-level abstractions that match the underlying mathematical concepts and generate the local assembly kernels from a symbolic representation of the PDE (in its variational form). In addition to Firedrake, recent successful examples in this context include FEniCS [Logg et al., 2012] and FreeFem++ [Hecht, 2012]. These software packages provide users with high-level interfaces for fast implementation while relying on optimisations and transformations in the code generation pipeline to generate efficient low-level code. The challenge, as in all compilers, is to create appropriate abstraction layers and apply optimisations that improve, or at least not worsen, performance on a broad set of programs and hardware platforms.

2.3 Firedrake overview and internal abstractions

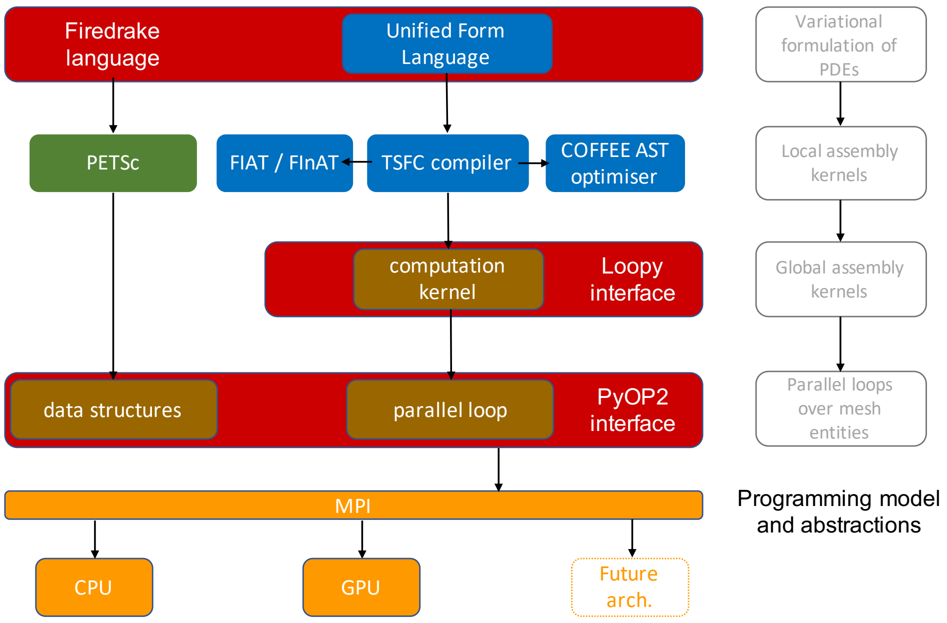

Figure 2.1: Simplified architecture diagram of the Firedrake software stack and its abstraction layers. Components shared with FEniCS are coloured in blue. Interfaces are coloured in red.

In this section, we give an overview of the software architecture of Firedrake, where the contributions described in this thesis are implemented and released. Firedrake [Rathgeber et al., 2015] is an automated system for the numerical solution of PDEs using the finite element method. One of the main design goals of Firedrake is to allow users to define the problems flexibly and intuitively while achieving high performance "out of the box". This goal motivates the design of the Firedrake system as various components that are organised into a multi-layered software stack with well-defined abstractions, providing the separation of concerns to domain specialists at each abstraction layer, as shown in Figure 2.1. We describe some of the components that are relevant to the work in this thesis below.

UFL

Firedrake and FEniCS [Logg et al., 2012] include the Unified Form Language (UFL) [Alnæs et al., 2014] as the high-level interface in Python for the users to define the variational formulation of the PDEs. In addition to UFL, Firedrake also exposes Python constructs and functions for tasks such as reading meshes from files, defining boundary conditions, specifying discretisation schemes and configuring the numerical linear algebra solvers. Firedrake relies on the DMPlex module [Knepley and Karpeev, 2009, Lange et al., 2016] in PETSc for distributed mesh management.

TSFC

The Two-Stage Form Compiler (TSFC) [Homolya et al., 2018] is a form compiler in Firedrake that automatically generates local assembly code for evaluating finite element integrals derived from the given variational forms. TSFC uses FIAT [Kirby, 2004] and FInAT [Homolya et al., 2017] to obtain the tabulation tensors of basis functions (and their derivatives) at specified quadrature points for the numerical computation of the integrals. Internally, TSFC defines an intermediate representation named GEM to describe the local assembly computation using symbolic tensor algebra. TSFC apply performance optimisations to this GEM representation by rewriting the tensor expressions before generating low-level code.

COFFEE

COFFEE [Luporini et al., 2015] is a domain-specific compiler and optimiser tailored to local assembly kernels for finite element methods. COFFEE analyses and transforms the abstract syntax tree (AST) of a local assembly kernel by exploiting particular characteristics of finite element integrals and achieves performance optimisations that are difficult for general-purpose compilers.

Loopy

Loopy [Klöckner, 2014] is a code generation system embedded in Python that is specialised for array-style computations. Internally, Loopy represents the computation domains, the data access patterns and the instruction dependencies as integer sets and relations. Firedrake leverages this powerful intermediate representation to describe and optimise the global assembly computation when evaluating finite element integrals, as described in Chapter 4.

PyOP2

PyOP2 [Rathgeber et al., 2012] is a Python library that provides the abstraction for specifying parallel iterations on entities of a general mesh. Started as a Python implementation of OP2 [Mudalige et al., 2012], PyOP2 shares the same basic concepts but also differs by using run-time code generation and just-in-time compilation. In Firedrake, PyOP2 organises the execution of the global assembly computation, coordinating data movement and synchronisation among compute nodes running in parallel.

2.4 Chapter summary

In this chapter, we present the essential mathematical concepts of the finite element method and how they are related to the main building blocks of Firedrake, with the emphasis on the computational properties of the local and global assembly processes. In the following chapters, we will discuss the design of the abstractions in Firedrake to support optimisations applied to the local and global assembly computations.

Chapter 3

Local assembly as tensor contractions

This chapter describes the computational aspects of the local assembly process in a typical finite element application. We argue that a symbolic, mathematical abstraction focused on tensor contractions is a suitable medium for representing the local assembly computation and applying certain optimisations. We lift loop optimisations in Firedrake, previously implemented by manipulating ASTs, to symbolic tensor computation rewrites in this higher level intermediate representation, and demonstrate its robustness and effectiveness with experimental results.

3.1 Local assembly as tensor contraction

We have illustrated in section 2.1.3 that the local assembly algorithms can be expressed as tensor contractions of GEM tensors in Firedrake. Since GEM nodes are purely symbolic, this representation enables robust expression rewrites following simple tensor algebra rules. The tensor contraction abstraction is an ideal layer to perform transformations aiming to reduce the number of floating-point operations, as it captures the mathematical specifics of the assembly algorithms while allowing freedom in the lower abstractions layers in realisation decisions such as instruction scheduling and hardware-specific optimisation such as SIMD vectorisation.

One benefit that the GEM presentation provides is the ability to rewrite the tensor computation efficiently. In local assembly routines such as Algorithm 3, often there are subexpressions in the inner loops which are invariant with respect to one or more of the outer iteration indices. The computation of these subexpressions can be hoisted outside of the invariant iterations by introducing temporary variables to store the results so that they are not recomputed unnecessarily. This technique is also called *loop-invariant code motion*, as demonstrated in works by Luporini et al. [2017] and Ølgaard and

```
for $k \leftarrow 1, \dots, N_k$ do
    for $j = 1, \dots, N_j$ do
        $\mathbf{A}_{jk} \leftarrow a_j \times c_k + a_j \times d_k$
    end for
end for
```

(a) The loop nest to compute $\mathbf{A}_{jk} = a_j c_k + a_j d_k$. Total number of floating-point operations is $3N_j N_k$.

```
for $k \leftarrow 1, \dots, N_k$ do
    $t \leftarrow c_k + d_k$
    for $j = 1, \dots, N_j$ do
        $\mathbf{A}_{jk} \leftarrow a_j \times t$
    end for
end for
```

(b) The loop nest to compute $\mathbf{A}_{jk} = a_j(c_k + d_k)$. Total number of floating-point operations is $N_k + N_k N_j$.

Figure 3.1: An example of loop-invariant code motion. The j-invariant subexpression $c_k + d_k$ can be lifted out of the j loop to reduce the amount of computation.

Wells [2010].

The effective application of loop-invariant code motion often requires rewriting the expression to expose invariant subexpressions. Consider the expression

$$A_{jk} = a_j c_k + a_j d_k \tag{3.1}$$

in a nested loop over j and k. In its original form, the subexpressions $a_j c_k$ and $a_j d_k$ are not invariant with respect to indices j and k, since both expressions depend on j and k. However, we can rewrite $a_j c_k + a_j d_k$ as $a_j(c_k + d_k)$, and the new expression $c_k + d_k$ is now invariant in j loop as it does not contain index j. This new subexpression can then be hoisted out into a temporary variable as a result. The pseudo-code generated by TSFC which illustrates the effect of loop-invariant code motion is shown in Figure 3.1.

We note that once an invariant subexpression is exposed by refactorisation, such as the example in Figure 3.1, the hoisting of the subexpression is performed automatically during code generation in TSFC. This is because the evaluation of GEM nodes during code generation is dictated by their free indices, as decried by Homolya [2018]. Furthermore, finite element assembly is only a specific subset of general tensor contraction computations, where we can leverage domain-specific information for optimisation. In particular, the bounds of all indices are known at compile time and because of the linearity of the multi-linear forms with respect to their arguments (i.e. the basis functions, this does not apply to the coefficients), each index to the arguments would only appear once in any GEM expression.

The main challenge to achieving effective loop-invariant code motion is that there are usually multiple ways to rewrite a tensor expression, and it is computationally infeasible to enumerate all the choices in general. As an example, consider the following expression

$$a_j c_k + a_j d_k + a_j e_k + b_j c_k + b_j e_k \tag{3.2}$$

Assuming indices j and k have the same extent, it is preferable to factorise it as $a_j(c_k+d_k+e_k)+b_j(c_k+e_k)$ rather than as $(a_j+b_j)c_k+a_jd_k+(a_j+b_j)e_k$ because the former requires less number of floating-point operations.

To summarise:

- Loop optimisations such as loop-invariant code motion are equivalent to targeted rewriting of GEM expressions using associativity and distributivity rules.
- Expansion and factorisation of expressions are often required to expose hoisting opportunities.
- Rewriting of expressions needs to be driven by effective cost models or heuristics leveraging domain-specific knowledge. Particularly to finite element methods, the knowledge that an expression is linear with respect to certain arguments can be leveraged to narrow down the rewrite search space, or as described later, make exhaustive searches feasible.

Historically, to optimise local assembly kernels in Firedrake, the COFFEE compiler [Luporini et al., 2015] is integrated into Firedrake. COFFEE creates an abstract syntax tree (AST) of the local assembly kernel and applies optimisations by systematically manipulating the AST. COFFEE applies both loop transformations, such as loop-invariant code motion, in order to reduce the number of floating-point operations, and also low-level optimisations, such as alignment and padding to promote vectorisation, to improve the execution efficiency of the hardware. One shortcoming of using COFFEE in Firedrake is that the AST representation is used both for optimisation and for code generation, and consequently, the optimisation passes need to be aware of code generation artefacts, such as declaration and initialisation of arrays, for correctness. Such considerations increase the complexity in the implementation of COFFEE. In addition, the representation of the computation as a tree instead of a graph makes it more challenging to discover hoistable subexpressions, as discussed later in this chapter.

COFFEE predates the development of TSFC as the default form compiler in Firedrake. TSFC brings GEM as an abstraction layer in the Firedrake software stack, which makes it possible to address some of the above issues by lifting the loop optimisation algorithms in COFFEE to a higher level abstraction based on symbolic mathematics.

3.2 Tensor expression rewrites in GEM

In this section, we describe the algorithm to rewrite the tensor expressions in GEM in order to reduce the number of floating-point operations.

3.2.1 Essential GEM concepts

We briefly describe some of the GEM constructs and concepts which are essential for our optimisation algorithm. This section only gives a much simplified view of GEM, and we refer the readers to [Homolya et al., 2018] for a thorough technical discussion.

DAGs Computations on tensors in GEM are organised as directed acyclic graphs (DAGs) of GEM objects. GEM objects form the nodes of the graph. No special meaning is assigned to the edges apart from indicating the "belongs to" relationship between nodes.

Tensors `Tensor` nodes represent multi-dimensional data in GEM. Constant tensors in GEM are known as `Literal`s, while tensors which are assigned values at runtime are known as `Variable`s. All tensor nodes in GEM have *shapes* which are known at compile-time.

Tensors $\leftrightarrow$ scalars Indexing a tensor with `Index` objects results in a scalar node. A `Tensor` node has a *shape* while an `Index` node has an *extent*. For example, if `T` is a `Variable` of shape $(3, 2)$, `i` is an `Index` of extent 3, `j` is an `Index` of extent 2, then `Indexed(T, (i, j))` is a scalar node representing the value `A[i, j]`.

Note that `i` and `j` are *free indices* in `A[i, j]`. They are lowered to loops during code generation. For instance, for the instruction `A[i, j]=0`, TSFC will generate nested loops which set all element of `A` to 0.

One can create `Tensor`s from scalar nodes with `ComponentTensor`. For example, `ComponentTensor(Indexed(T, (i, j)), (i,))` produces a `Tensor` with shape `(3,)` and free index `(j,)` that represents the slice `T[:, j]`.

Arithmetic operators GEM provides nodes for common arithmetic operations such as `Sum`, `Product`, and `MathFunction` (e.g. trigonometric functions). These operators only act on scalar arguments, thus tensor nodes need to be appropriatedly indexed beforehand.

Tensor contractions Tensor contractions are represented by `IndexSum` nodes in GEM. For instance, `IndexSum(e,i)` contracts expression `e` (which is assumed to have the free index `i`) over the summation index `i`.

Argument indices As explained in Section 2.1.3, a bilinear form $a(\cdot, \cdot)$ (or a multi-linear form in general) is always linear in its arguments. Since the local assembly matrix is defined as $\mathbf{A^K}_{ij} = a_K(\phi_i^K, \phi_j^K)$, we know that the resultant GEM expression is always linear with respect to terms indexed by i or j. We call these indices the *argument indices*. These are also the indices that appear on both sides of the assembly equation (i.e. they are not reduced over).

This linearity property guarantees that in any product node, each argument index appears at most once. As an example, in the product A_iB with i being an argument index, we can be sure that B is independent of i. Argument indices are therefore useful to expose loop-invariant subexpressions. The distinction between argument indices and other indices is always available at compile-time. For instance, in (2.29), the indices i and j are argument indices, while the indices q, α, β, γ are not.

Refactorisation labels GEM expressions can be assigned *refactorisation labels*. These labels determine how the expressions are handled in the rewriting passes. We define the following labels to support our algorithm:

- `ATOMIC`: the expression should not be broken up into smaller parts.
- `COMPOUND`: the expression can be broken up into smaller parts.
- `OTHER`: the expression is irrelevant for refactorisation.

Monomials A `Monomial` object represents a product of GEM nodes. This concept is introduced in GEM to facilitate the mechanics and analysis of refactoring product expressions. A `Monomial` is constructed from a list of *sum indices*, a list of GEM expressions called *atomics*, and another single GEM expression called *rest*. For instance, the expression $\sum_i a_1a_2r$ can be represented as

```
Monomial(sum_indices=(i,), atomics=(a1,a2), rest=r).
```

Note that the terms *atomics* and *rest* do not carry any semantic meaning at this stage: they simply specify the grouping of the factors in a product. The decision of whether an expression should be regarded as *atomic* or *rest* depends on specific optimisation passes.

Monomial sum A summation of `Monomials` are encapsulated in a `MonomialSum` object. In general, any finite element assembly integral can be written as the summation of products of tensors, or equally, as a `MonomialSum`.

3.2.2 Argument factorisation algorithm

We now describe the rewriting algorithm implemented in TSFC using the GEM abstraction to reduce the number of floating-point operations. Essentially, this algorithm is a reimplementation of certain optimisation passes in COFFEE [Luporini et al., 2017] in the language of tensor algebra. We name this algorithm *argument factorisation* because the key idea is to use arguments as common subexpressions for factorisation. Optimisations based on symbolic tensor algebra have been broadly explored in many projects, with Tensor Contraction Engine [Auer et al., 2006], TensorFlow [Abadi et al., 2016] and TVM [Chen et al., 2018] as some of the prominent examples in the applications of quantum chemistry and artificial neural network. Compared with these more general approaches, our algorithm differs by exploiting the linearity of some tensor elements, which is guaranteed by the finite element formulation, to arrive at simpler heuristics.

Since an integral that represents a finite element assembly computation is always linear with respect to its arguments, it is always possible to find factors in a product that are independent of some indices. Such factors can then be hoisted out of the inner loop(s) during code generation as loop-invariant subexpressions.

We use the Laplacian operator on a 2D triangular mesh to illustrate the argument factorisation algorithm. The bilinear form of this operator is given by

$$\mathbf{A}^K_{ij} = \sum_{q=1}^{N_q}\sum_{\alpha=1}^{2}\sum_{\beta=1}^{2}\sum_{\gamma=1}^{2} w_q \left({\mathbf{J}_\mathbf{K}^{-1}}_{\alpha\beta}\nabla\mathbf{\Phi}_{i\beta q}{\mathbf{J}_\mathbf{K}^{-1}}_{\alpha\gamma}\nabla\mathbf{\Phi}_{j\gamma q}\right)\left|\det(\mathbf{J}_\mathbf{K})\right|, \tag{3.3}$$

where the tensors are defined following the descriptions in Section 2.1.3. Note that since triangles are simplicial, the Jacobian is constant for each element and is independent of quadrature index q. Equation (3.3) is the input GEM expression to the argument factorisation algorithm.

Step 1. Expand the Jacobian applications and the dot products

Firstly, we expand the two Jacobian applications and the dot product in order to expose factorisation opportunities. This is achieved by unrolling any summation indices that have an extent less than or equal to 2 (the dimension of the physical space). In our example, the reductions over α, β, γ are unrolled to produce:

$$\mathbf{A}^K_{ij} = \sum_{q=1}^{N_q} w_q \left(P_{1iq}P_{1jq} + P_{2iq}P_{2jq}\right)\left|\det(\mathbf{J}_\mathbf{K})\right|. \tag{3.4}$$

where

$$P_{1jq} \equiv {\mathbf{J}_\mathbf{K}^{-1}}_{11}\nabla\mathbf{\Phi}_{jq1} + {\mathbf{J}_\mathbf{K}^{-1}}_{12}\nabla\mathbf{\Phi}_{jq2} \tag{3.5a}$$
$$P_{1kq} \equiv {\mathbf{J}_\mathbf{K}^{-1}}_{11}\nabla\mathbf{\Phi}_{kq1} + {\mathbf{J}_\mathbf{K}^{-1}}_{12}\nabla\mathbf{\Phi}_{kq2} \tag{3.5b}$$
$$P_{2jq} \equiv {\mathbf{J}_\mathbf{K}^{-1}}_{21}\nabla\mathbf{\Phi}_{jq1} + {\mathbf{J}_\mathbf{K}^{-1}}_{22}\nabla\mathbf{\Phi}_{jq2} \tag{3.5c}$$
$$P_{2kq} \equiv {\mathbf{J}_\mathbf{K}^{-1}}_{21}\nabla\mathbf{\Phi}_{kq1} + {\mathbf{J}_\mathbf{K}^{-1}}_{22}\nabla\mathbf{\Phi}_{kq2} \tag{3.5d}$$

Step 2. Convert expressions to `MonomialSums`

We introduce a classifier function that assigns refactorisation labels to GEM nodes that are factors of a `PRODUCT` node, according to the number of free indices and whether the free indices are argument indices (the set of which is known beforehand). This classifier is shown in Algorithm 5. We then recursively expand nodes that are labelled as `COMPOUND` by applying the distributive law $a(b+c) \rightarrow ab+ac$. In essence, this classifier expands all summations that contain more than one arguments and expose the `ATOMIC` nodes to the top level of the summation. Because the produced `ATOMIC` nodes tend to be shared among different terms in the summation, this expansion strategy is likely to introduce more refactorisation opportunities for the following steps.

Algorithm 5 Classifier to assign refactorisation labels to GEM nodes

```
function CLASSIFY(node, argument_indices)
    $n \leftarrow \left|\{\text{node.free\_indices}\} \cap \{\text{argument\_indices}\}\right|$
    if $n = 0$ then
        label ← OTHER
    else if $n = 1$ and node is Indexed then
        label ← ATOMIC
    else
        label ← COMPOUND
    end if
    return label
end function
```

The right-hand-side of Equation (3.3) is an `IndexSum` with a `Product` child node. The set of argument indices is $\{i, j\}$. The `Product` node has three child nodes. The child node $P_{0i}P_{0j} + P_{1i}P_{1j}$ is labelled as `COMPOUND` because it has two argument indices $\{i, j\}$, and therefore needs to be expanded by the distribution law. Note that a `COMPOUND` factor in a product must be a summation due to linearity. Thus such a rewrite is always possible. This produces the expression

$$w_q P_{1iq} P_{1jq} \left|\det(\mathbf{J_K})\right| + P_{2iq} P_{2jq} \left|\det(\mathbf{J_K})\right|. \tag{3.6}$$

This is a `Sum` node with two child `Product` nodes, and the above process repeats on the child `Product` nodes. The terms $P_{0i}, P_{0j}, P_{1i}, P_{1j}$ only have one argument index $\{i\}$ or $\{j\}$, but they are `Sum` nodes instead of `Indexed` nodes, thus they are labelled as `COMPOUND` and expanded as well. The terms w_q and $\left|\det(\mathbf{J_K})\right|$ are labelled as `OTHER` since they do not have any argument index.

After the expansion, all factors in each `Product` node are labelled either as `ATOMIC` or `OTHER`. We create a `Monomial` for each product, with the list of all `ATOMIC` factors as *atomics* and the product of all `OTHER` factors as *rest*. We then divide the `Monomials` into groups according to their *sum indices* and create a `MonomialSum` for each group.

There is only one sum index $\{q\}$ in (3.4), and the whole equation is converted into one `MonomialSum` consisting of four `Monomials`:

$$\mathbf{A}^K_{jk} = \overbrace{G_{11} + G_{21} + G_{12} + G_{22}}^{\texttt{MonomialSum}}, \tag{3.7}$$

where

$$G_{11} \equiv \overbrace{\underbrace{\sum_{q=1}^{N_q}}_{\texttt{sum index}} \underbrace{\nabla\mathbf{\Phi}_{jq1}\nabla\mathbf{\Phi}_{kq1}}_{\texttt{atomics}} \underbrace{\left(\mathbf{J}^{-1}_{\mathbf{K}\ 11}\mathbf{J}^{-1}_{\mathbf{K}\ 11} + \mathbf{J}^{-1}_{\mathbf{K}\ 21}\mathbf{J}^{-1}_{\mathbf{K}\ 21}\right) w_q \left|\det(\mathbf{J_K})\right|}_{\texttt{rest}}}^{\texttt{Monomial}} \tag{3.8a}$$

$$G_{21} \equiv \sum_{q=1}^{N_q} \nabla\mathbf{\Phi}_{jq2}\nabla\mathbf{\Phi}_{kq1} \left(\mathbf{J}^{-1}_{\mathbf{K}\ 11}\mathbf{J}^{-1}_{\mathbf{K}\ 12} + \mathbf{J}^{-1}_{\mathbf{K}\ 21}\mathbf{J}^{-1}_{\mathbf{K}\ 22}\right) w_q \left|\det(\mathbf{J_K})\right| \tag{3.8b}$$

$$G12 \equiv \sum_{q=1}^{N_q} \nabla\mathbf{\Phi}_{jq1}\nabla\mathbf{\Phi}_{kq2} \left(\mathbf{J}^{-1}_{\mathbf{K}\ 11}\mathbf{J}^{-1}_{\mathbf{K}\ 12} + \mathbf{J}^{-1}_{\mathbf{K}\ 21}\mathbf{J}^{-1}_{\mathbf{K}\ 22}\right) w_q \left|\det(\mathbf{J_K})\right| \tag{3.8c}$$

$$G_{22} \equiv \sum_{q=1}^{N_q} \nabla\mathbf{\Phi}_{jq2}\nabla\mathbf{\Phi}_{kq2} \left(\mathbf{J}^{-1}_{\mathbf{K}\ 12}\mathbf{J}^{-1}_{\mathbf{K}\ 12} + \mathbf{J}^{-1}_{\mathbf{K}\ 22}\mathbf{J}^{-1}_{\mathbf{K}\ 22}\right) w_q \left|\det(\mathbf{J_K})\right| \tag{3.8d}$$

Step 3. Pick optimal atomic factors to refactorise

For each `MonomialSum`, we define S_A as the set of all atomic factors. We choose the set $S_R \subseteq S_A$ consisting of *refactoring factors*. This subset is used to refactorise the summation. We required that each `Monomial` must have at least one atomic factor in S_R.

Algorithm 6 shows the recursive routine to find the set S_R. `Compare` is a function that ranks two possible sets of factors. In this case, we firstly favour solutions with fewer factors, as this means these factors are shared by more `Monomials`. In the cases of sets of the same size, we choose the set with factors whose argument indices have larger extents, because they correspond to loops with larger trip counts from which the invariant subexpressions could be hoisted out.

We then attempt to refactorise the `Monomials` with the factors in S_R, using the distributive rule $ab + ac \to a(b + c)$. This step produces a sum

Algorithm 6 Finding refactoring factors from all atomic factors.

```
$M \equiv \{\text{all monomials}\}$
$M_i \equiv i^{th}$monomial
$S_R \leftarrow \bigcup_{m \in M} \{\text{m.atomics}\}$                ▷ Initialise with set of all atomic factors
$S \leftarrow \{\}$

function COMPARE($S_1$, $S_2$)                      ▷ Rank two sets of atomic factors
    if $|S_1| < |S_2|$ then
        return True
    else if $|S_1| = |S_2|$ then
        $e_1 \leftarrow \sum_{a \in S_1} \prod_{i \in \text{a.ai}} (\text{i.extent})$   ▷ a.ai is the set of argument indices of a
        $e_2 \leftarrow \sum_{a \in S_2} \prod_{i \in \text{a.ai}} (\text{i.extent})$
        return $e_1 > e_2$
    else
        return False
    end if
end function

procedure FINDFACTORS($M$, $i$)                 ▷ Find factors for the $i^{th}$ Monomial
    if $i > |M|$ then
        return
    end if
    if $i = |M|$ then
        if COMPARE($S$, $S_R$) then
            $S_R \leftarrow S$                              ▷ Update the solution
        end if
        return
    end if
    for all $a \in \{M_i.\text{atomics}\}$ do
        $S \leftarrow S + a$                              ▷ Try each atomic factor
        FINDFACTORS($M$, $i + 1$)                    ▷ Go to the next monomial
        $S \leftarrow S - a$
    end for
end procedure

FINDFACTORS($M$, 1)                             ▷ Start from the first monomial
```

of products, with each product consisting of one `ATOMIC` factor and potentially another sum of products. The child `Sum` nodes are converted to new `MonomialSums` according to **Step 2**. We then repeat **Step 3** to the new `MonomialSums` recursively until they do not contain any `ATOMIC` factor. This

Original expression: $a_i b_{ij} e_{ik} z + d_i b_{ij} e_{ik} + c_i b_{ij} e_{ik} + a_q b_{ij} f_{ik} z$
Argument indices: $\{j, k\}$
Atomic factors: $S_A \equiv \{b_{ij}, e_{ik}, f_{ik}\}$
Refactoring factors: $S_R \equiv \{b_{ij}\}$
Refactorisation with S_R: $b_{ij} \underbrace{(a_i e_{ik} z + d_i e_{ik} + c_i e_{ik} + a_i f_{ik} z)}_{\text{new } \texttt{MonomialSum}}$
Recursively refactoring new `MonomialSum`:
  Atomic factors: $S'_A \equiv \{e_{ik}, f_{ik}\}$
  Refactoring factors: $S'_R \equiv \{e_{ik}, f_{ik}\}$
  Refactorisation with S'_R: $e_{ik}(a_i z + d_i + c_i) + f_{ik} a_i z$

Refactorisaiton result: $b_{ij} \underbrace{\left(e_{ik} \underbrace{(a_i z + d_i + c_i)}_{j,k\text{-invariant}} + f_{ik} a_i z \right)}_{j\text{-invariant}}$

Figure 3.2: An example of argument factorisation to expose loop-invariant subexpressions using Algorithm 6.

process is illustrated with an example in 3.2.

For the expression in (3.7), since the extents of j and k are equal, S_R can either take $S_{R_j} \equiv \{\nabla\mathbf{\Phi}_{jq1}, \nabla\mathbf{\Phi}_{jq2}\}$ or $S_{R_k} \equiv \{\nabla\mathbf{\Phi}_{kq1}, \nabla\mathbf{\Phi}_{kq2}\}$. Assuming $S_R = S_{R_j}$, we obtain the following expression:

$$\begin{aligned} \mathbf{A}^K_{jk} = &\sum_{q=1}^{N_q} \nabla\mathbf{\Phi}_{jq1} \left(\nabla\mathbf{\Phi}_{kq1} S_1 w_q \left|\det(\mathbf{J_K})\right| + \nabla\mathbf{\Phi}_{kq2} S_2 w_q \left|\det(\mathbf{J_K})\right|\right) + \\ &\sum_{q=1}^{N_q} \nabla\mathbf{\Phi}_{jq2} \left(\nabla\mathbf{\Phi}_{kq1} S_3 w_q \left|\det(\mathbf{J_K})\right| + \nabla\mathbf{\Phi}_{kq2} S_4 w_q \left|\det(\mathbf{J_K})\right|\right), \end{aligned} \tag{3.9}$$

with

$$S_1 \equiv \mathbf{J_K^{-1}}_{11}\mathbf{J_K^{-1}}_{11} + \mathbf{J_K^{-1}}_{21}\mathbf{J_K^{-1}}_{21} \tag{3.10a}$$
$$S_2 \equiv \mathbf{J_K^{-1}}_{11}\mathbf{J_K^{-1}}_{12} + \mathbf{J_K^{-1}}_{21}\mathbf{J_K^{-1}}_{22} \tag{3.10b}$$
$$S_3 \equiv \mathbf{J_K^{-1}}_{11}\mathbf{J_K^{-1}}_{12} + \mathbf{J_K^{-1}}_{21}\mathbf{J_K^{-1}}_{22} \tag{3.10c}$$
$$S_4 \equiv \mathbf{J_K^{-1}}_{12}\mathbf{J_K^{-1}}_{12} + \mathbf{J_K^{-1}}_{22}\mathbf{J_K^{-1}}_{22} \tag{3.10d}$$

We note that mathematically $S_2 = S_3$, but they correspond to different GEM nodes. Comparison between GEM expressions and trimming the DAG by folding equivalent nodes is a possible future improvement.

Step 4. Convert `MonomialSums` to expressions

Finally, we group the `Monomials` in each `MonomialSum` according to their sum indices, and create a `IndexSum` node for each group, using the *sum* *factorisation* algorithm described in [Homolya et al., 2017]. For arbitrary tensor contraction expressions, this algorithm searches for the optimal order of indices to contract with by traversing all the permutations of the contracting indices. This approach is computationally feasible because the number of sum indices is at most 3[1]. When handling each contraction index, we filter out the factors which need to be contracted and apply a greedy algorithm to determine the order of multiplication. The goal is to multiply the factors with free indices that have the smallest extents first, so that the inner loops have shorter trip counts. This process is shown in Algorithm 7.

This step converts Equation (3.9) to the following expression

$$\begin{aligned}\mathbf{A}^K_{jk} = &\sum_{q=1}^{N_q} \left(w_q \left|\det(\mathbf{J_K})\right| S_1 \nabla\mathbf{\Phi}_{kq1} + w_q \left|\det(\mathbf{J_K})\right| S_2 \nabla\mathbf{\Phi}_{kq2}\right) \nabla\mathbf{\Phi}_{jq1} + \\ &\sum_{q=1}^{N_q} \left(w_q \left|\det(\mathbf{J_K})\right| S_3 \nabla\mathbf{\Phi}_{kq1} + w_q \left|\det(\mathbf{J_K})\right| S_4 \nabla\mathbf{\Phi}_{kq2}\right) \nabla\mathbf{\Phi}_{jq2},\end{aligned} \tag{3.11}$$

In Listing 3.1, we show the C code generated by TSFC for the second-order Laplacian operator on a triangular mesh. This finite element discretisation has six basis functions in the local space. Because the integrand is a linear polynomial, the number of quadrature points is 3. The arrays `E1` and `E2` correspond to $\nabla\mathbf{\Phi_{iq1}}$ and $\nabla\mathbf{\Phi_{iq2}}$, the tabulations of the first derivatives of the basis functions in the x and y direction respectively. The array `w` holds the quadrature weights w_q.

Algorithm 7 Greedy algorithm to apply the associative rule to products in order to reduce the number of operations.

```
$i \equiv$ contraction index
$S_F \leftarrow$ {factors with index $i$}
**while** $|S_F| > 1$ **do**
                                        ▷ x.fi ≡ free indices of x
    $a, b \leftarrow \min_{a,b} \left\{ \prod_{j \in I} \text{j.extent} \middle| a, b \in S_F, a \neq b, I = \{\text{a.fi}\} \cup \{\text{b.fi}\} \right\}$
    $c \leftarrow a \times b$
    $S_F \leftarrow S_F \setminus \{a, b\}$
    $S_F \leftarrow S_F \cup \{c\}$
**end while**
```

[1] Since we always unroll the Jacobian applications and inner products, the only contractions are over quadrature indices. For simplicial elements, there is only one quadrature index. For tensor product finite elements, the number of quadrature indices is equal to the number of base elements that make up the tensor-product element. This number is bounded by the dimension of the physical space.

Listing 3.1: Local assembly kernel of the Laplace operator

```
static double const E1[3][6] = { ... };
static double const E2[3][6] = { ... };
static double const w[3] = { ... };

void kernel(double *__restrict__ A, double const *__restrict__ coords)
{
  // Jacobian
  double J11 = coords[2] - corrds[0];
  double J22 = coords[5] - coords[1];
  double J12 = coords[4] - coords[0];
  double J21 = coords[3] - corrds[1];
  double d = J11 * J22 - J12 * J21;
  double det = fabs(d);

  // Jacobian inverse
  double K11 = J22 / d;
  double K21 = - J21 / d;
  double K12 = - J12 / d;
  double K22 = J11 / d;

  double S1 = K11 * K11 + K21 * K21;
  double S2 = K11 * K12 + K22 * K21;
  double S3 = K11 * K12 + K21 * K22;
  double S4 = K21 * K21 + K22 * K22;

  double t0, t1, t2, t3, t4, t5[6], t6[6];
  for (int q = 0; q <= 2; ++q)
  {
    t0 = w[q] * det;
    t1 = t1 * S1;
    t2 = t1 * S2;
    t3 = t1 * S3;
    t4 = t1 * S4;
    for (int k = 0; k <= 5; ++k)
    {
      t4[k] = E1[q][k] * t1 + E2[q][k] * t2;
      t5[k] = E1[q][k] * t3 + E1[q][k] * t4;
    }
    for (int j = 0; j <= 5; ++j)
      for (int k_0 = 0; k_0 <= 5; ++k_0)
        A[j][k] += t4[k] * E1[q][j] + t5[k] * E2[q][j];
  }
}
```

3.3 Experimental evaluation

In this section, we assess the argument factorisation algorithm by applying it to a wide range of operators in Firedrake.

3.3.1 Experimental setup

TSFC organises the optimisation passes on GEM expressions as different *optimisation modes*, which the users can specify. Different modes share the same code generation backend but perform different GEM-to-GEM transformations. The simplest mode in TSFC is the `vanilla` mode, which does not attempt to refactorise the GEM expressions, i.e. it directly generates code from the GEM expressions produced by the frontend of the form compiler. This mode is the original behaviour of TSFC as described in [Homolya et al., 2018], and is the previous default in Firedrake.

Prior to the work described in this chapter, Firedrake relied on COFFEE [Luporini et al., 2017] to optimise the local assembly kernel. COFFEE represents the kernel as abstract syntax trees (ASTs) and performs transformations such as factorisation, expansion, association, code motion by manipulating the ASTs directly. Our approach described in this chapter is essentially a refinement of the ideas in COFFEE in an abstraction based on tensor algebra. This algorithm is organised as the `coffee` optimisation mode in TSFC. Raising the abstraction level enables us to perform the required transformations more robustly and efficiently: our implementation has less than 300 lines of additional Python code in TSFC. For comparison, the core components of COFFEE have more than 8000 lines of Python code. Part of the saving stems from the fact that scheduling and code generation are handled separately by TSFC after optimisation passes as applied, whereas COFFEE needs to rediscover the semantics of the computation and the dependencies between tensors. In addition, since the computation is represented as DAGs in GEM, a common subexpression stays as one GEM node throughout the rewrites, thus eliminating the need in COFFEE to perform analysis of the ASTs to rediscover common subexpressions. We note that COFFEE also performs other optimisations such as vectorisation, making it difficult to account for the lines of code in isolation. However, since the vectorisation pass is pushed down to the global assembly phase in Firedrake, as explained in Chapter 4, there are plans to retire COFFEE entirely from the Firedrake software stack in the near future.

This new setup for optimising local assembly kernels is schematically highlighted in Figure 3.3.

We evaluate the efficacy of our algorithm by measuring the compilation time (Section 3.4) and the number of floating-point operations (Section 3.5) of the local assembly kernels created from five bilinear forms: the mass matrix (`mass`), the Helmholtz equation (`helmholtz`), the vector Laplacian operator (`laplacian`), an elastic model (`elasticity`), and a hyperelastic model (`hyperelasticity`). The mathematical description of the operators is detailed in Appendix A.

We perform the experiments on 2D (triangles) and 3D (tetrahedra) meshes. We choose the Lagrange finite element of polynomial degree p for all the operators. We also multiply the bilinear form by c coefficient functions. The coefficient functions are from the Lagrange space of polynomial degree q. The ranges of the parameters in our test suite are:

- $p \in \{1, 2, 3, 4\}$

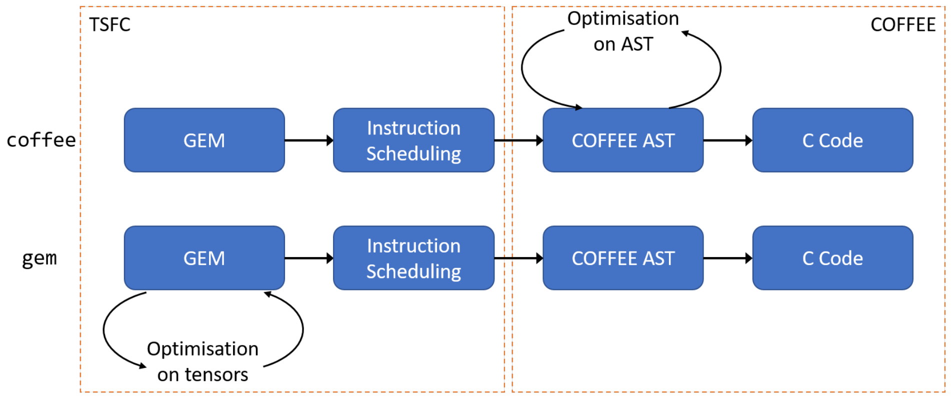

Figure 3.3: Comparison between the `coffee` and `gem` optimisation mode in TSFC. By raising the abstraction level, loop optimisations on ASTs can be lifted to transformations on tensors based on mathematical rules.

- $q \in \{1, 2, 3, 4\}$
- $c \in \{0, 1, 2, 3\}$

Our choice of operators and parameters are inspired by Ølgaard and Wells [2010] and Luporini et al. [2017], and are adapted from real-world applications. There are 640 test cases in total. We compare the results from three approaches:

- `vanilla:` Do not apply argument factorisation in TSFC and in COFFEE.
- `gem:` Apply the argument factorisation to the GEM expressions as described in this chapter[2].
- `COFFEE` Apply all COFFEE optimisations apart from vectorisation to the output of TSFC using `vanilla` mode. This is the previous default setting in Firedrake.

3.4 Compilation time

Firstly, we compare the time taken to perform the optimisations with `gem` and with `COFFEE`. Because both approaches act on the output of `vanilla`, we compute the time spent in the optimisation phase only by subtracting the compilation time of `vanilla` from the total compilation time. Other low-level optimisations in COFFEE, such as padding for promoting vectorisation, are switched off. The measurements are conducted on a system with

[2]Note that this pass is named the `coffee` mode in TSFC. We have modified the name here to avoid ambiguity when comparing with using COFFEE for optimisation.

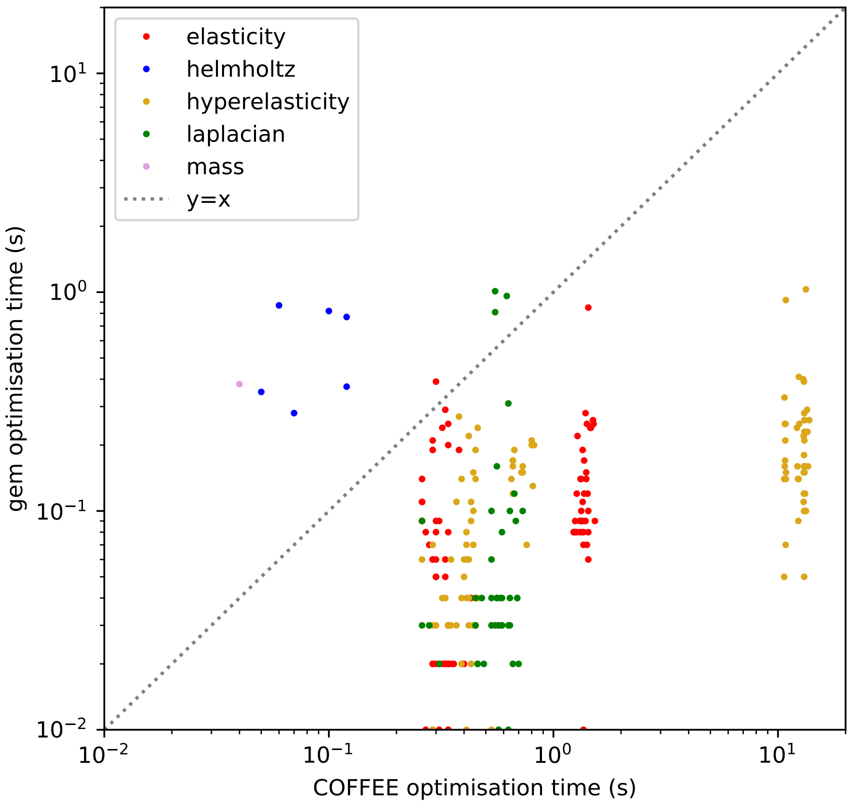

Figure 3.4: Comparison of time take to perform optimisations on the bilinear forms with `gem` and `COFFEE`. Each point represents one test case, placed at the coordinate {compilation time using `COFFEE`, compilation time using `gem`}. Only cases where the time taken is longer than 0.25 seconds are shown. The colours of the points distinguish different bilinear forms. The points above the $y = x$ line correspond to test cases where `gem` optimisations take longer than `COFFEE`, and vice versa. Optimisation with `gem` is faster than with `COFFEE` for 284 out of the 302 test cases shown.

an Intel Haswell CPU (Core i7-4790 3.60 GHz) with 32 GB DDR4 memory. Turbo Boost is disabled. We repeat the measurements three times and report the average time.

In Figure 3.4, we compare the time taken to perform argument factorisation in GEM (`gem`) and similar transformations in COFFEE (`COFFEE`). We show the results where either `gem` or `COFFEE` takes more than 0.25 seconds. Out of the 302 test cases, `gem` is faster than `COFFEE` in 284 cases.

The differences in optimisation time are more significant for complicated bilinear forms, such as `elasticity` and `hyperelasticity`. Such test cases have more complicated expressions and loop structures, resulting in more challenges in analysing and transforming the ASTs by the COFFEE compiler. The polynomial degree only changes the loop trip counts but not the expressions themselves. Therefore, we see that the compilation times for each bilinear form are clustered together.

3.5 Reducing floating-point operations

Next, we compare the efficacy of reducing the number of floating-point operations with `gem` and with `coffee`, using `vanilla` as the baseline. We define *optimisation effectiveness* as the reduction of the number of floating-point operations of the generated code against the `vanilla` strategy:

$$\text{optimisation effectiveness} = \frac{\texttt{vanilla}\ \ \text{FLOPs} - \texttt{coffee}\text{ or }\texttt{gem}\text{ FLOPs}}{\texttt{vanilla}\text{ FLOPs}}. \tag{3.12}$$

Figure 3.5 compares the optimisation effectiveness of the `coffee` approach and the `gem` approach. Out of the 640 test cases, the optimisation effectiveness differ by less than 5% in 466 cases, indicating that the argument factorisation algorithm is successful in performing the same optimisations by the COFFEE compiler. For all the remaining 174 test cases, the effectiveness of `gem` is higher than `coffee` by more than 5%. This result shows that while the argument factorisation approach is algorithmically similar to the optimisations in COFFEE, implementing the algorithms in the higher level of abstraction, based on tensor algebra, is more robust and thorough, resulting in fewer corner cases and unexpected performance degradations.

We list the exact experimental results for the `hyperelasticity` operator with two coefficients (i.e. $c = 2$) in Table 3.1.

Part of this advantage can be attributed to the intrinsic properties of the representation in GEM. For instance, GEM represents tensor contraction as DAGs instead of trees so that any common subexpressions are more likely to be kept intact after expanding a summation node. As an example, when rewriting $ab(c + d) \to abc + abd$, the product ab is still one node in the transformed DAG, and it only needs to be computed once in the generated code, whereas COFFEE will need to recover this kind of information after rewriting the ASTs.

In addition, by raising the abstraction level, optimisations in GEM are purely symbolic and mathematical. These optimisations are focused on operation count reduction, and in particular, are not concerned with instruction scheduling and code generation, which are handled separately afterwards. On the other hand, similar algorithms implemented in COFFEE, based on rewriting ASTs, need to take into account such semantics and tend to be

Table 3.1: Comparison of optimisation performances of `gem` and `COFFEE` for the `hyperelasticity` local assembly kernels with two coefficient functions. **D**: 2D or 3D mesh. **P**: Polynomial degree of the argument functions. **Q**: Polynomial degree of the coefficient functions. **flop**: Number of floating-point operations. **time**: Compilation time (in seconds). **E**: optimisation effectiveness as defined in (3.12), in percentage.

			`vanilla`		`COFFEE`			`gem`		
D	P	Q	flop	time	flop	E (%)	time	flop	E (%)	time
2	1	1	1,003	0.11	885	11.8	0.25	639	36.3	0.14
2	1	2	15,801	0.17	9,427	40.3	0.43	7,042	55.4	0.19
2	1	3	148,695	0.24	77,924	47.6	0.44	57,996	61.0	0.27
2	1	4	582,949	0.12	283,024	51.4	0.38	219,468	62.4	0.39
2	2	1	1,527	0.12	1,120	26.7	0.34	952	37.7	0.15
2	2	2	32,862	0.11	20,178	38.6	0.42	15,163	53.9	0.15
2	2	3	218,070	0.17	116,370	46.6	0.46	86,923	60.1	0.21
2	2	4	768,466	0.43	377,106	50.9	0.38	292,755	61.9	0.19
2	3	1	2,571	0.11	1,846	28.2	0.33	1,678	34.7	0.15
2	3	2	70,043	0.17	43,618	37.7	0.45	33,169	52.6	0.2
2	3	3	299,947	0.13	161,522	46.1	0.44	121,441	59.5	0.28
2	3	4	977,769	0.36	482,454	50.7	0.43	375,697	61.6	0.38
2	4	1	5,235	0.12	3,874	26.0	0.37	3,706	29.2	0.15
2	4	2	103,734	0.12	65,682	36.7	0.43	50,635	51.2	0.21
2	4	3	396,882	0.13	216,082	45.6	0.45	163,731	58.7	0.27
2	4	4	1,215,118	0.19	603,618	50.3	0.45	471,819	61.2	0.38
3	1	1	6,617	0.2	6,150	7.1	0.66	3,898	41.1	0.37
3	1	2	377,860	0.36	210,680	44.2	12.28	133,379	64.7	0.5
3	1	3	11,229,001	0.33	4,094,927	63.5	12.38	3,406,127	69.7	0.74
3	1	4	86,300,984	1.28	27,303,788	68.4	12.32	25,414,026	70.6	1.37
3	2	1	9,437	0.37	6,661	29.4	0.76	5,429	42.5	0.44
3	2	2	657,091	0.26	422,769	35.7	13.09	236,060	64.1	0.54
3	2	3	19,488,883	0.61	7,633,377	60.8	13.1	5,953,676	69.5	0.82
3	2	4	129,024,579	1.46	42,078,361	67.4	13.04	38,096,964	70.5	1.58
3	3	1	22,757	0.51	15,681	31.1	0.8	14,449	36.5	0.44
3	3	2	3,467,067	0.48	2,246,278	35.2	13.06	1,274,193	63.2	0.58
3	3	3	31,071,065	0.59	12,244,910	60.6	13.05	9,577,657	69.2	0.98
3	3	4	183,971,236	1.75	60,174,729	67.3	13.47	54,505,940	70.4	2.04
3	4	1	47,597	0.31	36,221	23.9	0.81	34,989	26.5	0.44
3	4	2	6,107,683	0.49	3,998,097	34.5	13.2	2,318,396	62.0	0.61
3	4	3	46,656,579	0.91	18,554,521	60.2	13.06	14,573,124	68.8	1.09
3	4	4	252,901,067	1.66	83,084,153	67.1	13.54	75,308,068	70.2	1.89

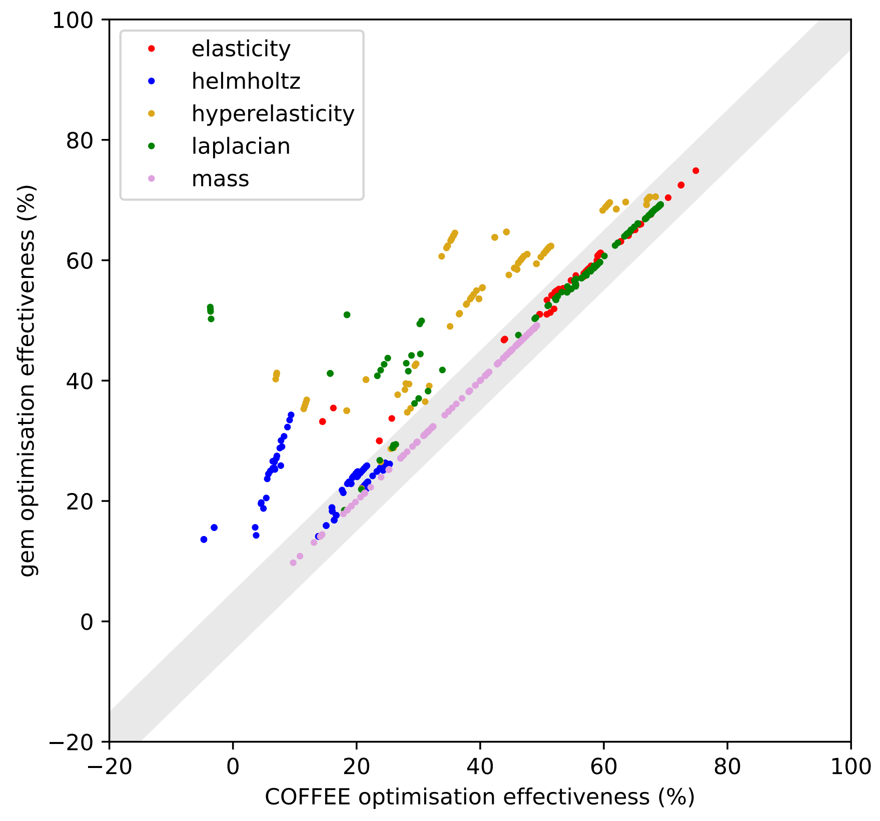

Figure 3.5: Comparison of optimisation effectiveness in reducing the number of floating-point operations reduction for `gem` and `COFFEE`. Each point represents one test case, placed at the coordinate {optimisation effectiveness with `COFFEE`, optimisation effectiveness with `gem`}. The colours of the points distinguish different bilinear forms. The shaded region represents the cases where the difference between the two approaches is less than 5%. Optimisation effectiveness with `gem` is higher than `COFFEE` by more than 5% for 174 out of the 640 test cases.

more mechanical in nature and more error-prone. For example, there have been a few instances where the declaration of a variable is moved after its uses, or a local variable is used out of its scope, leading to compilation errors.

Chapter 4

Vectorisation for global assembly of matrix-free operators

Global assembly in finite element methods is the process of computing the matrices and vectors representing the differential forms in the global function space. This is achieved by applying the local assembly kernel to each element in the domain and accumulate the local results. While optimisations of the local assembly kernels are focused on reducing the operation counts, which are largely hardware-oblivious, optimisations of the global assembly process are concerned with the efficient execution of the nested loops on specific hardware platforms. One particular optimisation of this nature is vectorisation on CPUs with SIMD (Single Instruction Multiple Data) instructions. In this chapter, we present the abstraction representing the global assembly process in Firedrake, and demonstrate how such an abstraction enables the loop transformations to generate vectorised code in a generic and automatic way.

4.1 Motivation and related works

One particular challenge for generating high-performance code on modern hardware is vectorisation. Modern CPUs increasingly rely on SIMD instructions to achieve higher throughput and better energy efficiency. Finite element computation requires the assembly of vectors and matrices, which represent differential forms on discretised function spaces. This process consists of applying a local function, often called an *element kernel*, to each mesh entity, and incrementing the global data structure with the local contributions. In the context of SIMD optimisation, the challenge is that typical local assembly kernels often suffer from issues that can preclude effective vectorisation. These issues include complicated loop structures, poor data access patterns, and loop trip counts that are not multiples of the vector width of the CPU. As we show later in this chapter, general purpose compilers frequently perform poorly in generating efficient, vectorised code for such kernels. For example, the loops in the local assembly kernel in Listing 4.2 are unlikely to be vectorised if the loop trip count (6) is not a multiple of the vector width, due to the cost of generating a remainder loop. In general, padding and data layout transformations are required to enable the vectorisation of the element kernels [Luporini et al., 2015], but the effectiveness of such approaches is not consistent across different examples. Since padding also results in larger overheads for wider vectors, new strategies are needed as vector width increases for the new generate of hardware.

Matrix-free methods, described in section 2.1.4, avoid building large sparse matrices in applications of the finite element method and thus trade computation for storage. They have become popular for use on modern hardware due to their higher arithmetic intensity (defined as the number of floating-point operations per byte of data transfer). SIMD vectorisation is particularly important for computationally intensive high order methods, for which matrix-free methods are often applied. Previous work on improving vectorisation of matrix-free operator application, or equivalently, residual evaluation, mostly focuses on exposing library interfaces to the users. Kronbichler and Kormann [2017] first perform a change of basis from nodal points to quadrature points, and provide overloaded SIMD types for users to write a quadrature-point-wise expression for residual evaluation. However, since the transformation is done manually, new operators require manual reimplementation. Knepley and Terrel [2013] also transpose to quadrature-point basis but target GPUs instead. Both works vectorise by grouping elements into batches, either to match the SIMD vector length in CPUs or the shared memory capacity on GPUs. In contrast, Müthing et al. [2017] apply an intra-kernel vectorisation strategy and exploit the fact that, in 3D, evaluating both a scalar field and its three derivatives fills the four lanes of an AVX2 vector register (assuming the computation is in double precision). More recently, Kempf et al. [2018] target high order Discontinuous Galerkin (DG) methods on hexahedral meshes using automated code generation to search for vectorisation strategies, while taking advantage of the specific memory layout of the data. Moxey et al. [2020] focus on vectorisation strategies for the matrix-free evaluation of the Helmholtz operator on high-order spectral/hp finite element, implemented in the Nektar++ framework [Cantwell et al., 2015].

Closer to the Firedrake framework, Mudalige et al. [2016] have successfully implemented automated inter-kernel vectorisation in OP2, where vectorised code can be generated from a user kernel specified in the OP2 API with minimal modification. However, in the PyOP2 layer in Firedrake, the computational kernels are treated as opaque functions, making it difficult for PyOP2 to perform the required code transformations.

In this chapter, we present a generic and portable solution based on cross-element vectorisation. Our vectorisation strategy, implemented in Firedrake, is similar to that of Kronbichler and Kormann [2017] but is fully automated through code generation like that of Kempf et al. [2018] and Mudalige et al. [2016]. We extend the scope of code generation in Firedrake to incorporate the outer iteration over mesh entities and leverage Loopy [Klöckner, 2014], a loop code generator based loosely on the polyhedral model, to systematically apply a sequence of transformations that promote vectorisation by grouping mesh entities into batches so that each SIMD lane operates on one entity independently. This automated code generation mechanism enables us to explore the effectiveness of our techniques on operators spanning a wide range of complexity and systematically evaluate our methodology. Compared with an intra-kernel vectorisation strategy, this approach is conceptually well-defined, more portable, and produces more predictable performance. Our experimental evaluation demonstrates that the approach consistently achieves a high fraction of hardware peak performance while being fully transparent to the end users.

The contributions of this chapter are as follows:

- We present the design of a code transformation pipeline that permits the generation of high-performance, vectorised code on a broad class of finite element models.
- We demonstrate the implementability of the proposed pipeline by realising it in the Firedrake finite element framework.
- We provide a thorough evaluation of our code generation strategy and demonstrate that it achieves a substantial fraction of theoretical peak performance across a broad range of test cases and popular C compilers.

The rest of this chapter is arranged as follows. After reviewing the preliminaries of code generation for the finite element method in Section 4.2, we describe our implementation of cross-element vectorisation in Firedrake in Section 4.3. In Section 4.5, we demonstrate the effectiveness of our approach with experimental results.

4.2 Preliminaries

As described in Sections 2.1.3 and 2.1.4, The assembly of a linear form in Firedrake is treated as a two-step process: *local* assembly and *global* assembly. The rest of this section highlights the computational properties of these two steps with an example, and in doing so, introduces the components and concepts in Firedrake that are relevant to the implementation of cross-element vectorisation.

4.2.1 Local assembly and TSFC

Local assembly of linear forms is the evaluation of the integrals as defined by the weak form of the differential equation on each entity (cell or facet) of the mesh. In Firedrake, the users define the problem in *Unified Form Language* (UFL) [Alnæs et al., 2014] which captures the weak form and the function space discretisation. Then the *Two-Stage Form Compiler* (TSFC) [Homolya et al., 2018] takes this high-level, mathematical description and generates efficient C code. The intermediate representation of TSFC is a tensor algebra language called *GEM*, which supports various optimisations on the tensor operations, as mentioned in Chapter 3. As an example, consider the linear form of the weak form of the positive-definite Helmholtz operator:

$$L(u; v) = \int_\Omega \nabla u \cdot \nabla v + uv \, \mathrm{d}x, \tag{4.1}$$

Listing 4.1: Assembling the linear form of the Helmholtz operator in UFL.

```
mesh = UnitSquareMesh(10, 10)
V = FunctionSpace(mesh, "Lagrange", 2)
v = TestFunction(V)
u = Function(V)
L = (dot(grad(u), grad(v)) + u*v) * dx
result = assemble(L)
```

Listing 4.1 shows the UFL syntax to assemble the linear form L as the vector `result`, on a 10×10 triangulation of a unit square. We choose to use the second-order Lagrange element, commonly known as the `P2` element, as our approximation space. Listing 4.2 shows a C representation of this kernel generated by TSFC[1]. We note the following key features of this element kernel:

- The kernel takes three array arguments in this case: `coords` holds the coordinates of the current triangle, `w_0` holds u_i, the coefficients of u, and `A` stores the result.

- The first part of the kernel (lines 8–20) computes the inverse and the determinant of the Jacobian for the coordinate transformation from the reference element to the current element. This is required for *pulling back* the differential forms to the reference element. The Jacobian is constant for each triangle because the coordinate transformation is affine in this case. In the general case, the Jacobian is computed at every quadrature point.

[1]This TSFC-generated kernel is reformatted slightly for consistency with the PyOP2 generated kernels. The kernel function names generated by TSFC and PyOP2 are long and complicated due to name mangling in Firedrake. They are shortened to operator names such as `helmholtz` in the listings for readability.

Listing 4.2: Local assembly kernel for the Helmholtz operator (degree 2, on triangles) of Listing 4.1 in C generated by TSFC.

```c
static inline void helmholtz(double *__restrict__ A,
  double const *__restrict__ coords, double const *__restrict__ w_0)
{
  double const t13[6] = { ... };
  double const t14[6 * 6] = { ... };
  double const t15[6 * 6] = { ... };
  double const t16[6 * 6] = { ... };
  double t0 = -1.0 * coords[0];
  double t1 = t0 + coords[2];
  double t2 = -1.0 * coords[1];
  double t3 = t2 + coords[5];
  double t4 = t0 + coords[4];
  double t5 = t2 + coords[3];
  double t6 = t1 * t3 + -1.0 * t4 * t5;
  double t7 = 1.0 / t6;
  double t8 = t1 * t7;
  double t9 = t7 * -1.0 * t4;
  double t10 = t7 * -1.0 * t5;
  double t11 = t3 * t7;
  double t12 = fabs(t6);
  for (int ip = 0; ip <= 5; ++ip) {
    double t17 = 0.0;
    double t18 = 0.0;
    double t19 = 0.0;
    for (int i = 0; i <= 5; ++i) {
      t17 = t17 + t16[6 * ip + i] * w_0[i];
      t18 = t18 + t15[6 * ip + i] * w_0[i];
      t19 = t19 + t14[6 * ip + i] * w_0[i];
    }
    double t20 = t13[ip] * t12;
    double t21 = t11 * t19 + t10 * t18;
    double t22 = t9 * t19 + t8 * t18;
    double t23 = t20 * (t21 * t10 + t22 * t8);
    double t24 = t20 * (t21 * t11 + t22 * t9);
    double t25 = t20 * t17;
    for (int j = 0; j <= 5; ++j)
      A[j] = A[j] + t15[6 * ip + j] * t23 + t16[6 * ip + j] * t25 +
          t14[6 * ip + j] * t24;
  }
}
```

- The constant arrays `t13`, `t14`, `t15`, `t16` are the same for all elements. `t14` represents the tabulation of basis functions at quadrature points, `t15` and `t16` represent derivatives of basis functions at quadrature points across each spatial dimension, `t13` represents the quadrature weights.

- The `ip` loop iterates over the quadrature points, evaluating the integrand in (4.1) and summing to approximate the integral. The `i` and `j` loops iterate over the degrees of freedom performing a change of basis to values at quadrature points and then back to degrees of freedom when accumulating into the output array `A`. The extents of these loops depend on the integrals performed and the choice of function spaces respectively.

- TSFC performs a sequence of optimisation passes by rewriting the tensor operations following mathematical rules before generating the loop nests (see Homolya et al. [2017] and Chapter 3 for details). This is more powerful than a C compiler's *loop-invariant code motion* optimisation because, firstly, TSFC operates on the symbolic tensor expressions to explore different refactoring and reordering strategies that expose invariant sub-expressions, and secondly, non-scalar sub-expressions can also be extracted into temporary arrays to eliminate redundant computation in the loop nests. As a side effect of these transformations, the loop nests in the kernels are no longer perfectly nested, thus limiting the effectiveness of vectorisation if it is only applied to the innermost loops (as most C compilers attempt to do).

- After the optimisation and scheduling stage, the translation of tensor algebra to C in TSFC is a straightforward rewrite of tensor operations to loop nests. This process results in certain artefacts that are shown in Listing 4.2. For example, since there is no specific representation for subtractions in TSFC, negations are emitted as multiplication by `-1`. This can result in generated C code that is not as idiomatic as if written by hand. Firedrake relies on the backend C compilers o optimise such artefacts away, since readability is a secondary concern for the generated code.

4.2.2 Global assembly and PyOP2

During global assembly, the local contribution from each mesh entity, computed by the element kernel, is accumulated into the global data structure. In Firedrake, PyOP2 [Rathgeber et al., 2012] is responsible for representing and realising the iteration over mesh entities, marshalling data in and out of the element kernels. The computation is organised as PyOP2 parallel loops, or *parloops*. A parloop specifies a computational kernel, a set of mesh entities to which the kernel is applied, and all data required for the kernel. The data objects can be directly defined on the mesh entities, or indirectly accessed through maps from the mesh entities. For instance, the signature for the global assembly of the Helmholtz operator is:

```
parloop(helmholtz, cells, L(cell2dof, INC), coords(cell2vert, READ),
        u(cell2dof, READ)).
```

Here `helmholtz` is the element kernel as shown in Listing 4.2, generated by TSFC; `cells` is the set of all triangles in the mesh; `L`, `coords`, and `u` are the global data objects that are needed to create the arguments for the element kernel, where `L` holds the result vector, `coords` holds the coordinates of the vertices of the triangles which are needed for computing the Jacobian, and `u` holds the vector representation of function u (as weights of basis functions). These global data objects correspond to the kernel arguments `A`, `coords` and `w_0` respectively. The maps `cell2dof` and `cell2vert` provide indirections from the mesh entities to the global data objects. Finally, each data argument is annotated with an access descriptor: `READ` for read-only, `INC` for increment access. In this example, the `L` and `u` arguments share the same map (since they are both defined on the same quadratic Lagrange space), while the `coords` argument, being linear, uses a different map.

Listing 4.3 shows the C code generated by PyOP2 for the above example. The code is then JIT-compiled when the result is needed in Firedrake. In the context of vectorisation, this approach, with the inlined element kernel, forms the baseline in our experimental evaluation. We note the following key features of the global assembly kernels:

- The outer loop is over mesh entities.
- For each entity, the computation can be divided into three parts: gathering the input data from global into local data (`t3` and `t4` in this case, which correspond to kernel arguments `coords` and `w_0`), calling the local assembly kernel, and scattering the output data (`t2`) to the global data structure.
- The gathering and scattering of data make use of indirect addressing via base pointers (`dats`) and indices (`maps`).
- Different mesh entities might share the same degrees of freedom: parallelisation of the scattering loop on line 29 must be aware of the potential for data races.
- Global assembly interacts with local assembly via a function call (line 27). While the C compiler can potentially inline this call, it creates an artificial boundary to using loop optimisation techniques that operate at the source code level. Additionally, even after inlining, outer loop vectorisation over mesh entities requires that the C compiler vectorises

Listing 4.3: Global assembly code for action of the Helmholtz operator (degree 2, on triangles) in C generated by PyOP2.

```c
static inline void helmholtz(double *__restrict__ A,
  double const *__restrict__ coords, double const *__restrict__ w_0)
{
  // ... element kernel as defined previously ... //
}

void wrap_helmholtz(int const start, int const end,
  double *__restrict__ dat0, double const *__restrict__ dat1,
  double const *__restrict__ dat2, int const *__restrict__ map0,
  int const *__restrict__ map1)
{
  double t2[6];
  double t3[3 * 2];
  double t4[6];

  for (int n = start; n <= -1 + end; ++n) {
    for (int i6 = 0; i6 <= 5; ++i6)
      t4[i6] = dat2[map0[6 * n + i6]];

    for (int i2 = 0; i2 <= 2; ++i2)
      for (int i3 = 0; i3 <= 1; ++i3)
        t3[2 * i2 + i3] = dat1[2 * map1[3 * n + i2] + i3];

    for (int i1 = 0; i1 <= 5; ++i1)
      t2[i1] = 0.0;

    helmholtz(t2, t3, t4);

    for (int i15 = 0; i15 <= 5; ++i15)
      dat0[map0[6 * n + i15]] += t2[i15];
  }
}
```

through data-dependent array accesses. This is the software engineering challenge that has previously limited vectorisation to a single local assembly kernel in Firedrake.

4.3 Vectorisation

As one would expect, the loop nests and loop trip counts vary considerably for different integrals, meshes and function spaces that users might choose. This complexity is one of the challenges that our system specifically, and Firedrake more generally, must face in order to deliver predictable performance on modern CPUs, which have increasingly rich SIMD instruction sets.

In the prior approach to vectorisation in our framework, the local assembly kernels generated by TSFC were further transformed to facilitate vectorisation, as described in Luporini et al. [2015]. The arrays are padded so that the trip counts of the innermost loops match multiples of the length of SIMD units. However, padding becomes less effective for low polynomial degrees on wide SIMD units. For instance, AVX512 instructions act on 8 double-precision floats, but the loops for degree 1 polynomials on triangles only have trip counts of 3. Moreover, loop-invariant code motion is very effective in reducing the number of floating-point operations, but the hoisted instructions are not easily vectorised as they are no longer in the innermost loops. This effect is more pronounced on tensor-product elements where TSFC is able to apply *sum factorisation* [Homolya et al., 2017] to achieve better algorithmic complexity.

4.3.1 Cross-element vectorisation and Loopy

Another strategy is to vectorise across several elements in the outer loop over the mesh entities, as proposed previously by Kronbichler and Kormann [2017]. This approach computes the contributions from several mesh entities using SIMD instructions, where each SIMD lane handles one entity. This is always possible regardless of the complexity of the local element kernel because the computation on each entity is independent and identical. One potential downside is the increase in register and cache pressure as the working set is larger.

For a compiler, the difficulty in performing cross-element vectorisation (or, more generally, outer-loop vectorisation) is to automate a sequence of loop transformations and necessary data layout transformations robustly. This is further complicated by the indirect memory accesses in data gathering and scattering, and the need to unroll and interchange loops across these indirections. Such transformations require significantly more semantic knowledge than what is available to the C compiler.

Loopy [Klöckner, 2014] is a loop generator embedded in Python which targets both CPUs and GPUs. Loopy provides abstractions based on integer sets for loop-based computations and enables powerful transformations based on the polyhedral model [Verdoolaege, 2010]. Loop-based computations in Loopy are represented as *Loopy kernels*. A Loopy kernel is a subprogram consisting of a loop domain and a partially-ordered list of scalar assignments acting on multi-dimensional arrays. The loop domain is specified as a set of integral points in the convex intersection of quasi-affine constraints, represented using the Integer Set Library [Verdoolaege, 2010]. Loopy supports code generation for different environments from the same kernel by choosing different *targets*.

To integrate with Loopy, the code generation mechanisms in Firedrake were modified as illustrated in Figure 4.1.

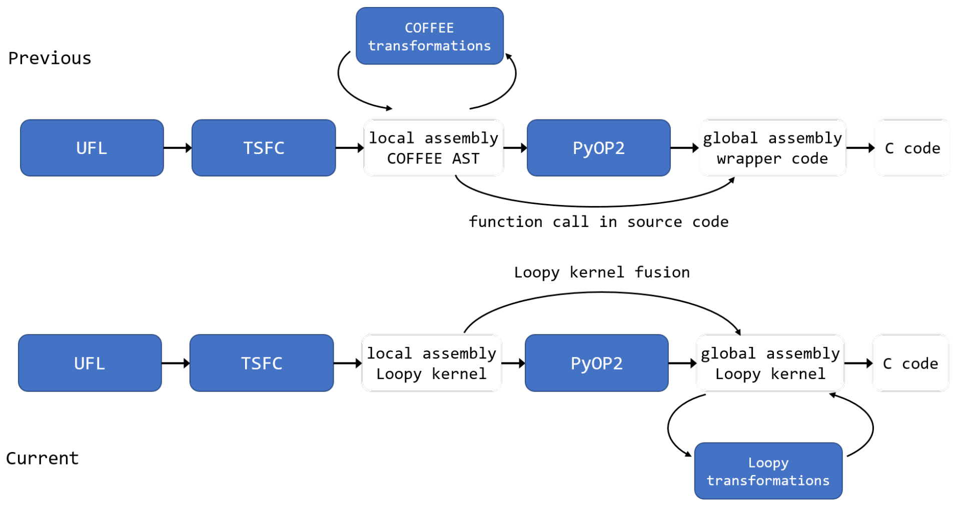
Previous
COFFEE transformations
UFL
TSFC
local assembly COFFEE AST
PyOP2
global assembly wrapper code
C code
function call in source code
Loopy kernel fusion
UFL
TSFC
local assembly Loopy kernel
PyOP2
global assembly Loopy kernel
C code
Loopy transformations
Current

Figure 4.1: Integration of Loopy in Firedrake for global assembly code generation.

Instead of generating source code directly, TSFC and PyOP2 are modified to generate Loopy kernels. We have augmented the Loopy internal representation with the ability to support a generalised notion of kernel fusion through the nested composition of kernels, specifically through subprograms and inlining. This allows PyOP2 to inline the element kernel such that the global assembly Loopy kernel encapsulates the complete computation of global assembly. This holistic view of the overall computation enables robust loop transformations for vectorisation across the boundary between global and local assembly. To facilitate SIMD instruction generation, we also introduced a new OpenMP target to Loopy which extends its existing C-language target to support OpenMP SIMD directives [OpenMP Architecture Review Board, 2018, §2.9.3].

Listing 4.4 shows an abridged version of the global assembly Loopy kernel for the Helmholtz operator, with the element kernel fused.

Listing 4.4: Global assembly Loopy kernel of the Helmholtz operator (degree 2, on triangles).

```
KERNEL: helmholtz
---------------------------------------------------------------------
ARGUMENTS:
start: type: int32
end: type: int32
dat0: type: float64, shape: (None)
// ... More arguments ... //
---------------------------------------------------------------------
DOMAINS:
[end, start] -> { [n] : start <= n < end }
{ [i6] : 0 <= i6 <= 5 }
// ... More domains ... //
---------------------------------------------------------------------
INAME_IMPLEMENTATION_TAGS:
None
---------------------------------------------------------------------
TEMPORARIES:
t4: type: float64, shape: (6), dim_tags: (stride:1)
// ... More temporaries ... //
---------------------------------------------------------------------
INSTRUCTIONS:
for n
  for i6
    t4[i6] = dat2[map0[n, i6]]
  // ... More instructions ... //
  for i15
    dat0[map0[n, i15]] += t0[0, i15]
end n
```

We highlight the following key features of Loopy kernels:

- Loop indices, such as `n` and `i1`, are called *inames* in Loopy, which define the iteration space. The bounds of the loops are specified by the affine constraints in *domains*.

- Loop transformations operate on kernels by rewriting the loop domain and the statements making up the kernel. In addition, each iname carries a set of *tags* governing its realisation in generated code, perhaps as a sequential loop, as a vector lane index, or through unrolling.

- Multi-dimensional arrays occur as *arguments* and *temporaries*. The memory layout of the data can be specified by assigning *tags* to the array dimensions.

- Dependencies between statements determine their partial order. Statement scheduling can also be controlled by assigning priorities to statements and inames.

For example, to achieve cross-element vectorisation (by batching four elements into one SIMD vector in this example) we invoke the following sequence of Loopy transformations on the global assembly Loopy kernel, exploiting the domain knowledge of finite element assembly:

1. Split the outer loop `n` over mesh entities into `n_outer` and `n_simd`, with `n_simd` having a trip count of four. The objective is to generate SIMD instructions for the `n_simd` loops, such that each vector lane computes one iteration of the `n_simd` loops.
2. Assign the tag `SIMD` to the new iname `n_simd`. This tag informs Loopy to force the `n_simd` loop to be innermost, privatising the temporary variables by vector-expansion if necessary.

We show the change to the Loopy kernel after these transformations in Listing 4.5. In particular, the vector-expansion of the temporary `t4`, and the splitting (and subsequent modification of the loop domain) of the `n` iname.

Listing 4.5: Changes to global assembly Loopy kernel of the Helmholtz operator after cross-element vectorisation

```
KERNEL: helmholtz_simd
---------------------------------------------------------------------
ARGUMENTS:
start: type: int32
end: type: int32
dat0: type: float64, shape: (None)
// ... More arguments ... //
---------------------------------------------------------------------
DOMAINS:
[end, start] -> { [n_outer, n_simd] :
  n_simd >= start - 4n_outer and
  0 <= n_simd <= 3 and n_simd < end - 4n_outer }
// ... More domains ... //
---------------------------------------------------------------------
INAME_IMPLEMENTATION_TAGS:
n_simd: SIMD
---------------------------------------------------------------------
TEMPORARIES:
t4: type: float64, shape: (6, 4), dim_tags: (stride:4, stride:1)
// ... More temporaries ... //
---------------------------------------------------------------------
INSTRUCTIONS:
for n_outer, n_simd
  for i6
    t4[i6, n_simd] = dat2[map0[n_outer * 4 + n_simd, i6]]
  // ... More instructions ... //
  for i15
    dat0[map0[n_outer * 4 + n_simd, i15]] += t2[i15, n_simd]
end n_outer, n_simd
```

Listing 4.6 shows the generated C code for the Helmholtz operator vectorised by grouping together four elements. Apart from the previously mentioned changes, we note the following details:

Listing 4.6: Global assembly code for action of the Helmholtz operator (degree 2, on triangles) in C vectorised by batching four elements.

```c
// ... Constant array declarations ... //

void wrap_helmholtz(int const start, int const end,
  double *__restrict__ dat0, double const *__restrict__ dat1,
  double const *__restrict__ dat2, int const *__restrict__ map0,
  int const *__restrict__ map1)
{
  double form_t1[4] __attribute__ ((aligned (64)));
  double t2[6 * 4] __attribute__ ((aligned (64)));
  // ... More temporary array declarations ... //
  for (int n_outer = (start / 4); n_outer < (end / 4); ++n_outer) {
    for (int i2 = 0; i2 <= 2; ++i2) {
      for (int i3 = 0; i3 <= 1; ++i3) {
        #pragma omp simd
        for (int n_simd = 0; n_simd <= 3; ++n_simd)
          t3[n_simd + 8 * i2 + 4 * i3] =
                dat1[2 * map1[12 * n_outer + 3 * n_simd + i2] + i3];
      }
    }
    for (int i6 = 0; i6 <= 5; ++i6) {
      #pragma omp simd
      for (int n_simd = 0; n_simd <= 3; ++n_simd)
        t4[n_simd + 4 * i6] =
          dat2[map0[24 * n_outer + 6 * n_simd + i6]];
    }
    for (int i1 = 0; i1 <= 5; ++i1) {
      #pragma omp simd
      for (int n_simd = 0; n_simd <= 3; ++n_simd)
        t2[n_simd + 4 * i1] = 0.0;
    }
    #pragma omp simd
    for (int n_simd = 0; n_simd <= 3; ++n_simd) {
      form_t11[n_simd] = 0.0;
      form_t1[n_simd] = -1.0 * t3[n_simd];
      // ... More similar instructions ... //
      form_t8[n_simd] = fabs(form_t7[n_simd]);
    }
    for (int form_ip = 0; form_ip <= 5; ++form_ip) {
      // ... More similar loop nests ... //
      for (int form_j = 0; form_j <= 5; ++form_j) {
        #pragma omp simd
        for (int n_simd = 0; n_simd <= 3; ++n_simd)
          t2[n_simd + 4 * form_j] +=
            form_t25[form_j] * form_t24[n_simd] +
            form_t13[n_simd + 4 * form_j] +
            form_t27[form_j] * form_t26[n_simd];
      }
    }
    for (int i15 = 0; i15 <= 5; ++i15)
      for (int n_simd = 0; n_simd <= 3; ++n_simd)
        dat0[map0[24 * n_outer + 6 * n_simd + i15]] +=
          t2[n_simd + 4 * i15];
  }
}
```

- The `n_simd` loops are pushed to the innermost level. Moreover, this transformation vector-expands temporary arrays such as `t2`, `t3`, `t4` by four, with the expanded dimension labelled as varying the fastest when viewed from (linear) system memory. This ensures their accesses in the `n_simd` loops always have unit stride.

- Loopy provides a mechanism to declare arrays to be aligned to specified memory boundaries (64 bytes in this example).
- The `n_simd` loops are decorated with `#pragma omp simd` to inform C compilers to generate SIMD instructions. The exception is the writing back to the global array (lines 49–52), which is sequentialised due to potential race conditions, as different mesh entities could share the same degrees of freedom.
- The peel loops that handle the cases where the number of elements is non-divisible by four are generated by specifying *slabs* in Loopy. These loops process the first and final iterations, which are potentially incomplete, so that the conditionals in bounds of the main loop are simplified. These peel loops are omitted here for simplicity. Note that it is possible to pass additional constraints to Loopy if more information is known about the loop bounds in PyOp2, which enables Loopy to eliminate these loops in certain cases.
- After cross-element vectorisation, all local assembly instructions (lines 31–48) are inside `n_simd` loops, which always have trip counts of four and are unit stride. All loop-varying array accesses are unit stride in the fastest moving dimension. There are no loop-carried dependencies in `n_simd` loops. As a result, the `n_simd` loops, and therefore all local assembly instructions, are vectorisable without further consideration of dependencies. This is verified by checking the x86 assembly code and also by running the program with the Intel Software Development Emulator.

4.4 Compiler vector extensions

A more direct way to inform the compiler to emit SIMD instructions without depending on OpenMP is to use *vector extensions*[2], which support vector data types. These were first introduced in the GNU compiler (GCC), but are also supported in recent versions of the Intel C compiler (ICC) and Clang. Analogous mechanisms exist in various vector-type libraries, e.g. VCL [Fog, 2017]. To evaluate and compare with the directive-based approach from Section 4.3.1, we created a further code generation target in Loopy to support vector data types. When inames and corresponding array axes are jointly tagged as vector loops, Loopy generates code to compute on data in vector registers directly, instead of scalar loops over the vector lanes. It is worth

[2]https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html

noting that the initial intermediate representation of the loop is identical in each case, and that the different specialisations are achieved through code transformation.

Listing 4.7: Global assembly code for action of the Helmholtz operator in C vectorized by four elements (using vector extensions).

```c
typedef double double4 __attribute__ ((vector_size (32)));
typedef int int4 __attribute__ ((vector_size (16)));

static double4 const _zeros_double4 __attribute__ ((aligned (64))) =
    { 0.0 };

// ... Constant array declarations ... //

void wrap_helmholtz(int const start, int const end,
  double *__restrict__ dat0, double const *__restrict__ dat1,
  double const *__restrict__ dat2, int const *__restrict__ map0,
  int const *__restrict__ map1)
{
  double4 form_t1 __attribute__ ((aligned (64)));
  // ... Temporary array declarations ... //

  for (int n_outer = (start / 4); n_outer <= ((-4 + end) / 4);
       ++n_outer) {
    for (int i2 = 0; i2 <= 2; ++i2) {
      for (int i3 = 0; i3 <= 1; ++i3) {
        #pragma omp simd
        for (int n_simd = 0; n_simd <= 3; ++n_simd)
          t3[n_simd + 8 * i2 + 4 * i3] =
            dat1[2 * map1[12 * n_outer + 3 * n_simd + i2] + i3];
      }
    }
    for (int i6 = 0; i6 <= 5; ++i6) {
      #pragma omp simd
      for (int n_simd = 0; n_simd <= 3; ++n_simd)
        t4[n_simd + 4 * i6] =
          dat2[map0[24 * n_outer + 6 * n_simd + i6]];
    }

    for (int i1 = 0; i1 <= 5; ++i1)
      t2[i1] = _zeros_double4;

    form_t11 = _zeros_double4;
    form_t1 = -1.0 * t3[0];

    for (int form_ip = 0; form_ip <= 5; ++form_ip) {
      // ... More similar instructions ... //
      #pragma omp simd
      for (int n_simd = 0; n_simd <= 3; ++n_simd)
        form_t8[n_simd] = fabs(form_t7[n_simd]);
      // ... More similar instructions ... //
      for (int form_j = 0; form_j <= 5; ++form_j)
        t2[form_j] +=
          form_t25[form_j] * form_t24 +
          form_t13[form_j] +
          form_t27[form_j] * form_t26;
    }

    for (int i15 = 0; i15 <= 5; ++i15)
      for (int n_simd = 0; n_simd <= 3; ++n_simd)
        dat0[map0[24 * n_outer + 6 * n_simd + i15]] +=
            t2[i15][n_simd];
  }
}
```

Listing 4.7 shows the C code generated for the Helmholtz operator vectorised by batching four elements using the vector extension target. Here almost all vectorised (innermost) loops for local assembly are replaced by operations on vector variables. For instructions which do not fit the vector computation model, most noticeably the indirect data gathering (lines 22–24 29–31), or instructions containing built-in mathematics functions which are not supported on vector data types (line 44), Loopy generates scalar loops over vector lanes, decorated with `#pragma omp simd`. In addition, because vector extensions do not automatically broadcast scalars, any vector instruction with a scalar rvalue is modified by adding the zero vector to the expression, as shown in lines 35 and 37.

Compared to Listing 4.6, using vector extensions removes most of the innermost loops, and the only remaining OpenMP SIMD directives are due to the limitations of vector extensions as explained previously.

4.5 Performance Evaluation

We follow the performance evaluation methodology of [Luporini et al., 2017] by measuring the assembly time of a range of operators of increasing complexity and polynomial degrees. Due to the large number of combinations of experimental parameters (operators, meshes, polynomial degrees, vectorisation strategies, compilers, hyperthreading), we only report an illustrative portion of the results here, with the entire suite of experiments made available on the interactive online repository CodeOcean [Sun, 2019b].

4.5.1 Experimental setup

We performed experiments on a single node of two Intel systems, based on the Haswell and Skylake microarchitectures, as detailed in Table 4.1. Firedrake uses MPI for parallel execution where each MPI process handles the assembly for a subset of the domain. Hybrid MPI-OpenMP parallelisation is not supported, and we stress that OpenMP pragmas are only used for SIMD vectorisation within a single MPI process. Because we observe that hyperthreading usually improves the performance by 5% to 10% for our applications, we set the number of MPI processes to the number of *logical* cores of the CPU to utilise all available computational resources. Experimental results with hyperthreading turned off are available on CodeOcean. Turbo Boost is switched off to mitigate reproducibility problems that might be caused by dynamic thermal throttling. The batch size, i.e., the number of elements grouped together for vectorisation, is chosen to be consistent with the SIMD length. We use three C compilers: GCC 7.3, ICC 18.0 and Clang

Table 4.1: Hardware specifications for experiments on cross-element vectorisation

	Haswell Xeon E5-2640 v3	**Skylake Xeon Gold 6130**
Base frequency	2.6 GHz	2.1 GHz
Physical cores	8	16
SIMD instruction set	AVX2	AVX512
`doubles` per SIMD vector	4	8
Cross-element vectorization batch size	4	8
FMA[3] units per core	2	2
FMA instruction issue per cycle	2	2
Peak performance (double-precision)[4]	332.8 GFLOP/s	1075.2 GFLOP/s
System memory	4×8 GB DDR4-2133	2×32 GB DDR4-2666
LINPACK performance (double-precision)[5]	262.5 GFLOP/s	678.8 GFLOP/s
Memory bandwidth[6]	38.5 GB/s	36.6 GB/s
GCC/Clang arch flag	-march=native	-march=native
ICC SIMD flag	-xcore-avx2	-xcore-avx512 -qopt-zmm-usage=high
Other compiler flags	-O3 -ffast-math -fopenmp	-O3 -ffast-math -fopenmp
Intel Turbo Boost	OFF	OFF

5.0. The two vectorisation strategies described in Section 4.3 are tested on all platforms. We use the vendor-published *Base Frequency* to calculate the peak performance in Table 4.1. In reality, modern Intel CPUs dynamically reduce frequencies on heavy workloads with AVX2 and AVX512 instructions, which results in lower achievable performance. To get a reasonable indication of the *achievable* peak performance for compute-bound applications, we run the optimised LINPACK benchmark binary provided by Intel and record its performance on the two platforms.

For the benefit of reproducibility, we have archived the specific versions of Firedrake components that are used for the experimental evaluation on Zenodo [Zenodo/Firedrake, 2019]. An installation of Firedrake with components matching the ones used in this chapter can be obtained following the instruction at https://www.firedrakeproject.org/download.html, using the following command:

```
python3 firedrake-install --doi 10.5281/zenodo.2595487
```

The evaluation framework itself, including the program formulations and scripts for performance measurement, is archived at [Sun, 2019a].

[3]Fused multiply-add operations.

[4]Calculated as *base frequency* × *#cores* × *SIMD width* × 2 *(for FMA)* × *#issue per cycle*

[5]Intel LINPACK Benchmark. *https://software.intel.com/en-us/articles/intel-mkl-benchmarks-suite*

[6]STREAM triad benchmark, 2 threads per core.

Table 4.2: Operator characteristics and speed-up summary, using GCC with vector extensions. AI: arithmetic intensity (FLOP/byte). D: trip count of loops over degrees of freedom. Q: trip count of loops over quadrature points. H: speed-up over baseline on Haswell, 16 processes, with vector extensions. S: speed-up over baseline on Skylake, 32 processes, with vector extensions.

		`tri`					`quad`					`tet`					`hex`				
	P	AI	D	Q	H	S	AI	D	Q	H	S	AI	D	Q	H	S	AI	D	Q	H	S
mass	1	1.2	3	3	1.0	1.0	4.7	2	3	1.1	1.5	2.7	4	4	1.2	0.7	16.9	2	3	1.8	2.8
	2	1.7	6	6	1.3	1.0	3.9	3	4	0.8	1.0	5.9	10	14	1.7	2.4	10.8	3	4	1.1	1.5
	3	3.0	10	12	2.0	1.3	3.9	4	5	0.8	1.0	8.7	20	24	0.9	1.8	8.5	4	5	1.8	2.5
	4	5.6	15	25	2.4	2.6	3.9	5	6	2.2	1.9	39.2	35	125	1.0	1.6	7.4	5	6	2.1	2.8
	5	7.5	21	36	1.1	2.0	3.9	6	7	2.3	1.5	55.9	56	216	0.7	1.0	7.0	6	7	2.0	2.7
	6	9.7	28	49	0.8	1.6	4.1	7	8	2.5	1.9	81.2	84	343	1.1	1.9	6.9	7	8	2.2	2.7
helmholtz	1	1.8	3	3	1.2	1.0	10.7	2	3	2.0	2.9	3.9	4	4	1.6	1.6	45.5	2	3	2.5	3.5
	2	5.7	6	6	2.2	1.7	10.6	3	4	1.0	1.3	27.3	10	14	2.3	5.5	34.9	3	4	1.8	3.3
	3	9.6	10	12	2.3	5.6	10.5	4	5	1.3	2.1	37.5	20	24	1.5	3.3	27.9	4	5	1.8	3.2
	4	17.8	15	25	2.3	5.2	10.5	5	6	3.2	5.7	164.1	35	125	1.9	2.8	24.5	5	6	2.8	4.7
	5	23.3	21	36	1.7	3.5	10.4	6	7	2.8	4.8	230.1	56	216	1.4	1.8	23.1	6	7	2.8	4.5
	6	29.9	28	49	1.3	2.8	10.9	7	8	3.0	5.0	331.4	84	343	1.5	4.1	22.7	7	8	2.9	4.4
laplacian	1	0.5	3	1	1.0	1.1	7.9	2	3	1.7	2.7	1.9	4	1	1.0	1.0	37.8	2	3	2.3	3.2
	2	2.7	6	3	1.7	1.4	8.7	3	4	1.0	1.2	10.4	10	4	2.2	3.6	27.1	3	4	1.9	2.8
	3	4.0	10	6	2.2	2.1	8.4	4	5	1.5	2.0	24.0	20	14	2.2	3.5	21.6	4	5	1.5	2.0
	4	6.9	15	12	2.4	2.8	8.3	5	6	3.1	2.9	31.5	35	24	2.7	3.2	19.2	5	6	2.6	3.7
	5	12.6	21	25	2.1	3.1	8.2	6	7	2.9	3.9	124.3	56	125	2.9	2.6	18.4	6	7	2.5	3.7
	6	16.8	28	36	1.9	2.7	8.6	7	8	2.8	4.0	189.3	84	216	2.6	2.3	18.3	7	8	2.5	4.1
elasticity	1	0.5	3	1	1.0	1.0	10.2	2	3	1.9	2.8	1.9	4	1	1.1	1.0	48.0	2	3	2.3	3.3
	2	3.0	6	3	1.8	1.5	10.2	3	4	0.9	1.3	11.6	10	4	2.0	6.3	31.8	3	4	1.9	2.9
	3	4.4	10	6	2.1	2.2	9.5	4	5	1.5	2.0	25.6	20	14	2.2	3.3	24.4	4	5	1.5	1.9
	4	7.3	15	12	2.5	3.6	9.2	5	6	3.0	3.9	32.7	35	24	2.8	3.1	21.3	5	6	2.6	3.7
	5	13.2	21	25	2.1	3.2	9.0	6	7	2.8	3.9	127.5	56	125	2.8	2.6	20.1	6	7	2.5	3.8
	6	17.4	28	36	1.9	2.8	9.3	7	8	2.8	4.3	192.6	84	216	2.6	2.4	19.8	7	8	2.4	4.2
hyperelasticity	1	0.5	3	1	1.5	1.4	31.9	2	4	1.2	1.9	1.7	4	1	1.6	1.9	183.0	2	4	1.2	1.9
	2	9.8	6	6	2.6	4.2	31.3	3	6	1.9	3.0	63.2	10	14	3.1	5.9	137.8	3	6	2.3	4.3
	3	26.8	10	25	3.1	6.7	30.6	4	8	1.4	1.6	313.8	20	125	3.0	6.1	118.6	4	8	1.5	1.7
	4	40.7	15	49	3.3	7.3	30.6	5	10	3.3	6.3	600.0	35	343	2.9	4.2	110.3	5	10	3.2	6.0
	5	56.1	21	81	2.6	6.1	30.5	6	12	3.4	6.6	915.2	56	729	2.6	2.8	108.1	6	12	3.0	5.5
	6	74.5	28	121	2.3	4.3	31.8	7	14	3.3	5.9	1428.3	84	1331	1.7	4.6	108.8	7	14	3.0	5.7

We measure the execution time of assembling the residual for five operators: the mass matrix ("`mass`"), the Helmholtz equation ("`helmholtz`"), the vector Laplacian ("`laplacian`"), an elastic model ("`elasticity`"), and a hyperelastic model ("`hyperelasticity`"). The mathematical description of the operators is detailed in Appendix A. These operators stem from real-world applications and cover a wide range of complexity: the generated C code for the corresponding global assembly kernels exceeds hundreds of KB for the `hyperelasticity` operator at high polynomial degree.

We performed experiments on both 2D and 3D domains, with two types of mesh used for each case: triangles ("`tri`") and quadrilaterals ("`quad`") for 2D problems, tetrahedra ("`tet`") and hexahedra ("`hex`") for 3D problems. This large variety underscores the broad applicability of our approach. The *arithmetic intensities* and other pertinent characteristics of the operators are listed in Table 4.2. The memory footprint is calculated assuming perfect caching – it is thus a lower bound which results in an upper bound estimation for the arithmetic intensity. The triangular and tetrahedral meshes use an affine coordinate transformation (requiring only one Jacobian evaluation per element). The quadrilateral and hexahedral meshes use a bilinear (trilinear) coordinate transformation (requiring Jacobian evaluation at every quadrature point), which usually results in higher arithmetic intensities at low orders. In Firedrake, tensor-product elements [McRae et al., 2014] benefit from optimisations such as sum factorisation to achieve lower asymptotic algorithmic complexity. They are therefore more competitive for higher order methods [Homolya et al., 2017].

We record the maximum execution time of the generated global assembly kernels on all MPI processes. This time does not include the time in synchronisation and MPI data exchange for halo updates. Each experiment is run five times, and the average execution time is reported. Exclusive access to the compute nodes is ensured and threads are pinned to individual logical cores. Startup costs such as code generation time and compilation time are excluded. We use automatic vectorisation by GCC without batching, compiled with the optimisation flags of Table 4.1, as the baseline for comparison. Compared with our cross-element strategy, the baseline represents the out-of-the-box performance of compiler auto-vectorisation for the local element kernel. We note that cross-element vectorisation does not alter the algorithm of local assembly except for the vector expansion, as illustrated by Listing 4.2 and Listing 4.6. Consequently, the total number of floating-point operations remains the same. The performance benefit from cross-element vectorisation is therefore composable with the operation-reduction optimisations performed by the form compiler to the local assembly kernels.

Before measuring the performance in terms of floating-point operations per second (FLOPs), we use the Intel Software Development Emulator (In-

Table 4.3: Statistics of floating-point arithmetic instructions for degree 4 `helmholtz` kernel on `quad` mesh, running on Haswell. The `Baseline` column and `Vect` columns show the numbers of instructions (in millions) executed without and with cross-element vectorisation. GCC is used for both cases, with the flags listed in Table 4.1. The `Vect FLOPs` column shows the contribution of FLOPs by different types of instructions. We consider one FMA instruction as two FLOPs (e.g. one AVX2 FMA instruction contributes eight FLOPs).

Instruction type	**Baseline**	**Vect**	**Vect FLOPs (%)**
1 double (scalar)	15,695	344	0.4
2 doubles (SSE)	3,962	0	0.0
4 doubles (AVX2)	2,372	4,621	21.6
FMA 1 double (scalar)	13,996	0	0.0
FMA 2 doubles (SSE)	655	0	0.0
FMA 4 doubles (AVX2)	3,041	8,349	78.0
Total	36,691	13,314	100.0

tel SDE)[7] to verify the effectiveness of our transformation in promoting vectorisation. We count the number of floating-point arithmetic instructions and group them by (A) their operand sizes and (B) whether they are fused multiply-add (FMA) instructions. As an example, Table 4.3 shows the statistics for the `helmholtz` operator of polynomial degree 4 on a `quad` mesh, running on Haswell. The data indicate that after cross-element vectorisation, the compiler is more capable of generating SIMD instructions for the global assembly. In this particular example, 99.6% of all floating-point computations are performed by AVX2 SIMD instructions, compared with less than 40% for the baseline.

4.5.2 Experimental results and discussion

Figures 4.2 to 4.5 show the performance of the operators on Haswell and Skylake, vectorised with OpenMP SIMD directives (Section 4.3.1), and with vector extensions (Section 4.4). We indicate the fraction of peak performance achieved on the left axis, and the fraction of the LINPACK benchmark performance on the right axis. Figures 4.6 and 4.7 compare the roofline models [Williams et al., 2009] of the baseline and our approach using vector extensions, using the GCC compiler on Haswell and Skylake. The speed-up achieved is also summarised in Table 4.2.

[7]https://software.intel.com/en-us/articles/intel-software-development-emulator

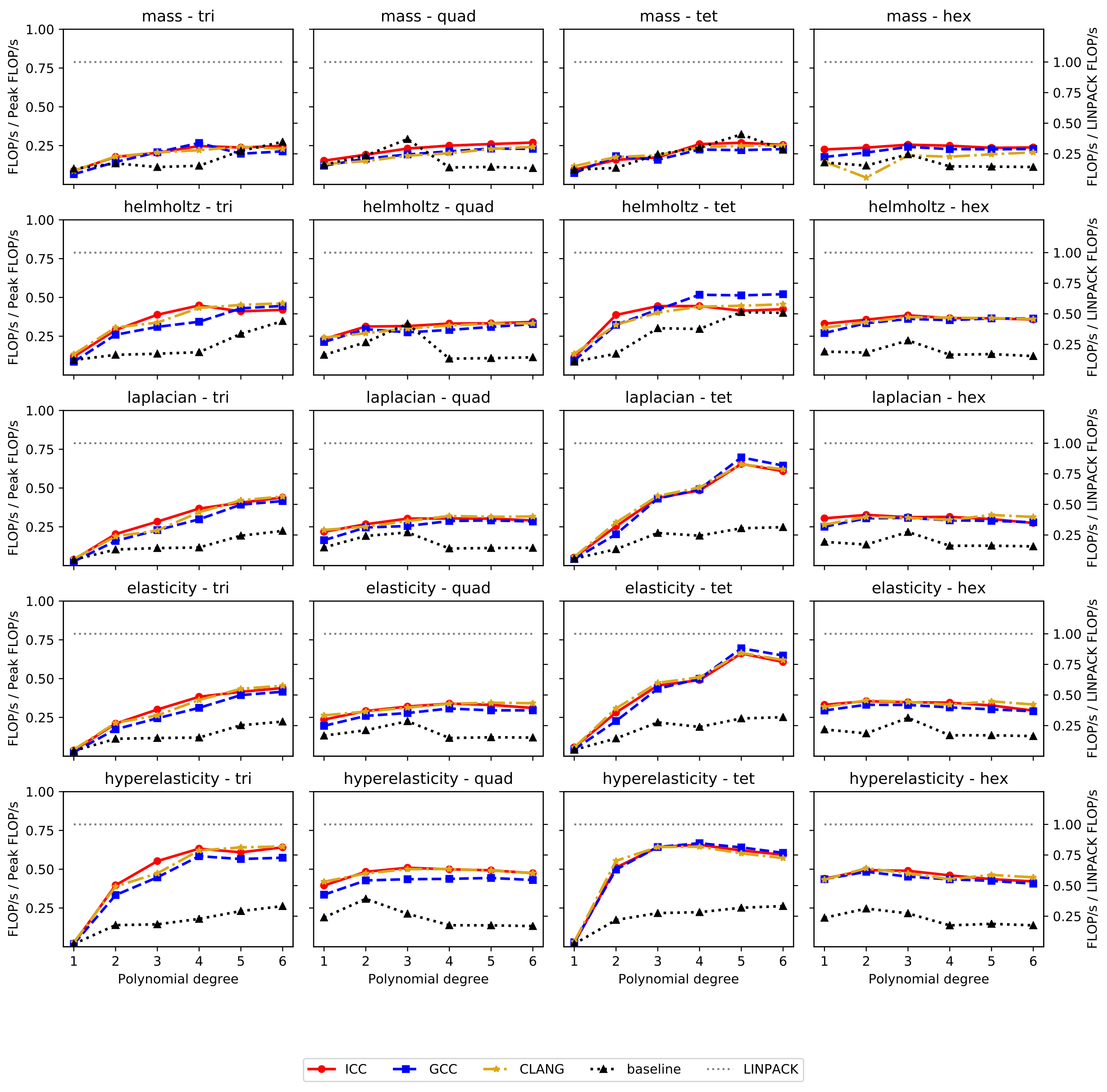

Figure 4.2: The fraction of peak FLOP/s (as listed in Table 4.1) achieved by different compilers for operators {`mass`, `helmholtz`, `laplacian`, `elasticity`, `hyperelasticity`}, on meshes {`tri`, `quad`, `tet`, `hex`} on **Haswell** using **vector extensions** with 16 MPI processes. The dotted line indicates the fraction of peak performance achieved by LINPACK benchmark.

.

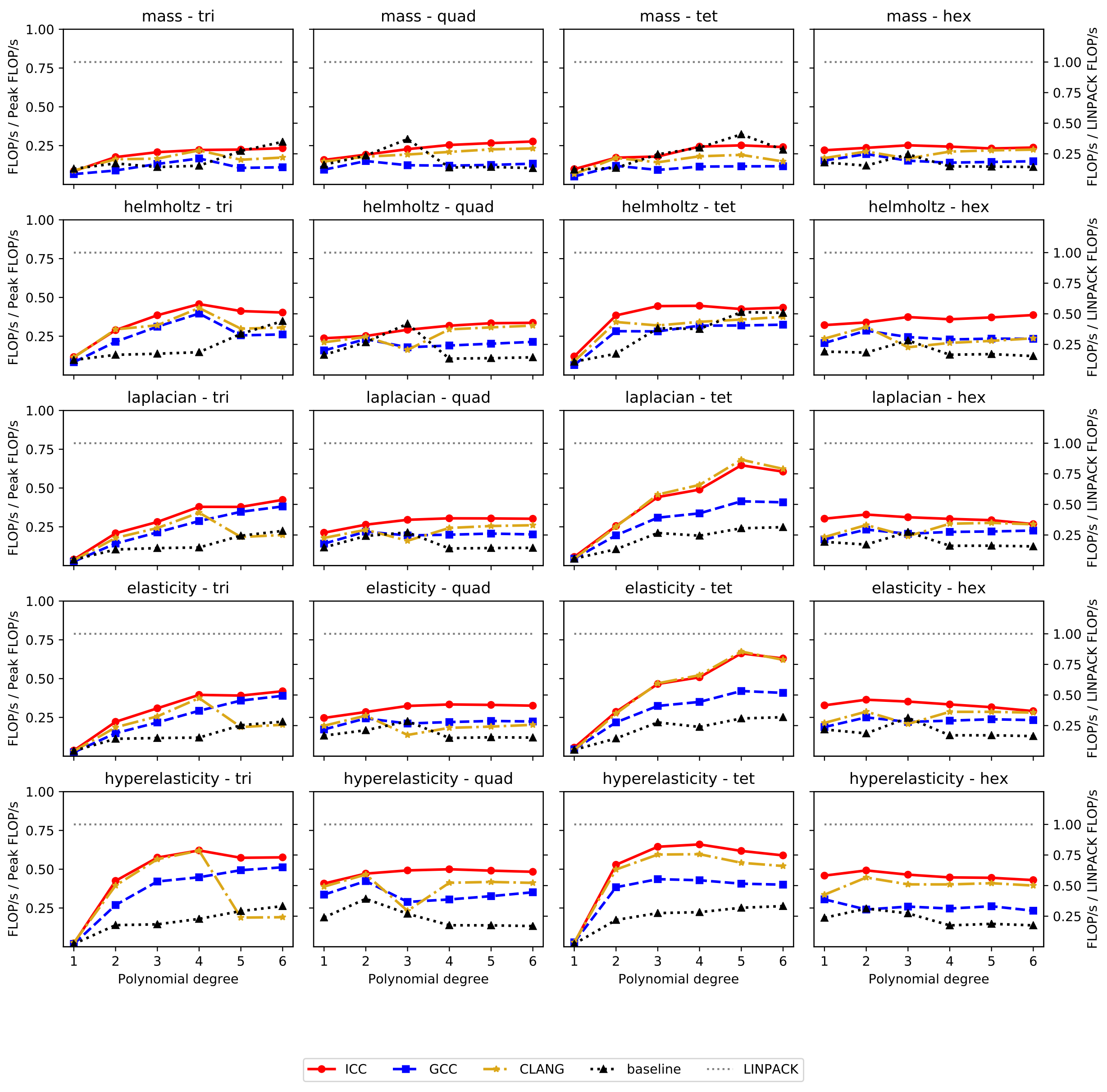

Figure 4.3: The fraction of peak FLOP/s (as listed in Table 4.1) achieved by different compilers for operators {`mass`, `helmholtz`, `laplacian`, `elasticity`, `hyperelasticity`}, on meshes {`tri`, `quad`, `tet`, `hex`} on **Haswell** using **OpenMP pragma** with 16 MPI processes. The dotted line indicates the fraction of peak performance achieved by LINPACK benchmark.

.

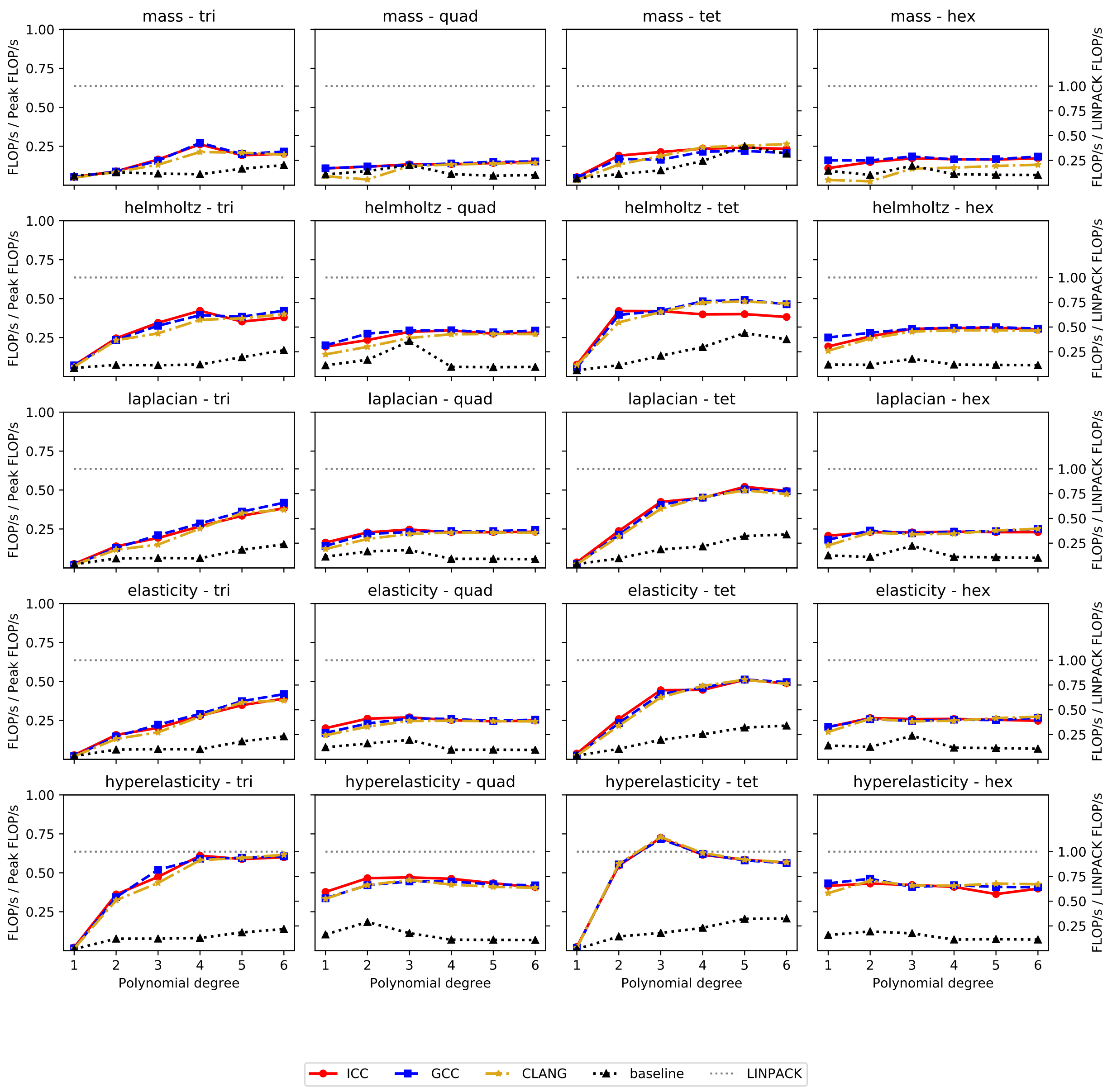

Figure 4.4: The fraction of peak FLOP/s (as listed in Table 4.1) achieved by different compilers for operators {`mass`, `helmholtz`, `laplacian`, `elasticity`, `hyperelasticity`}, on meshes {`tri`, `quad`, `tet`, `hex`} on **Skylake** using **vector extensions** with 32 MPI processes. The dotted line indicates the fraction of peak performance achieved by the LINPACK benchmark.

.

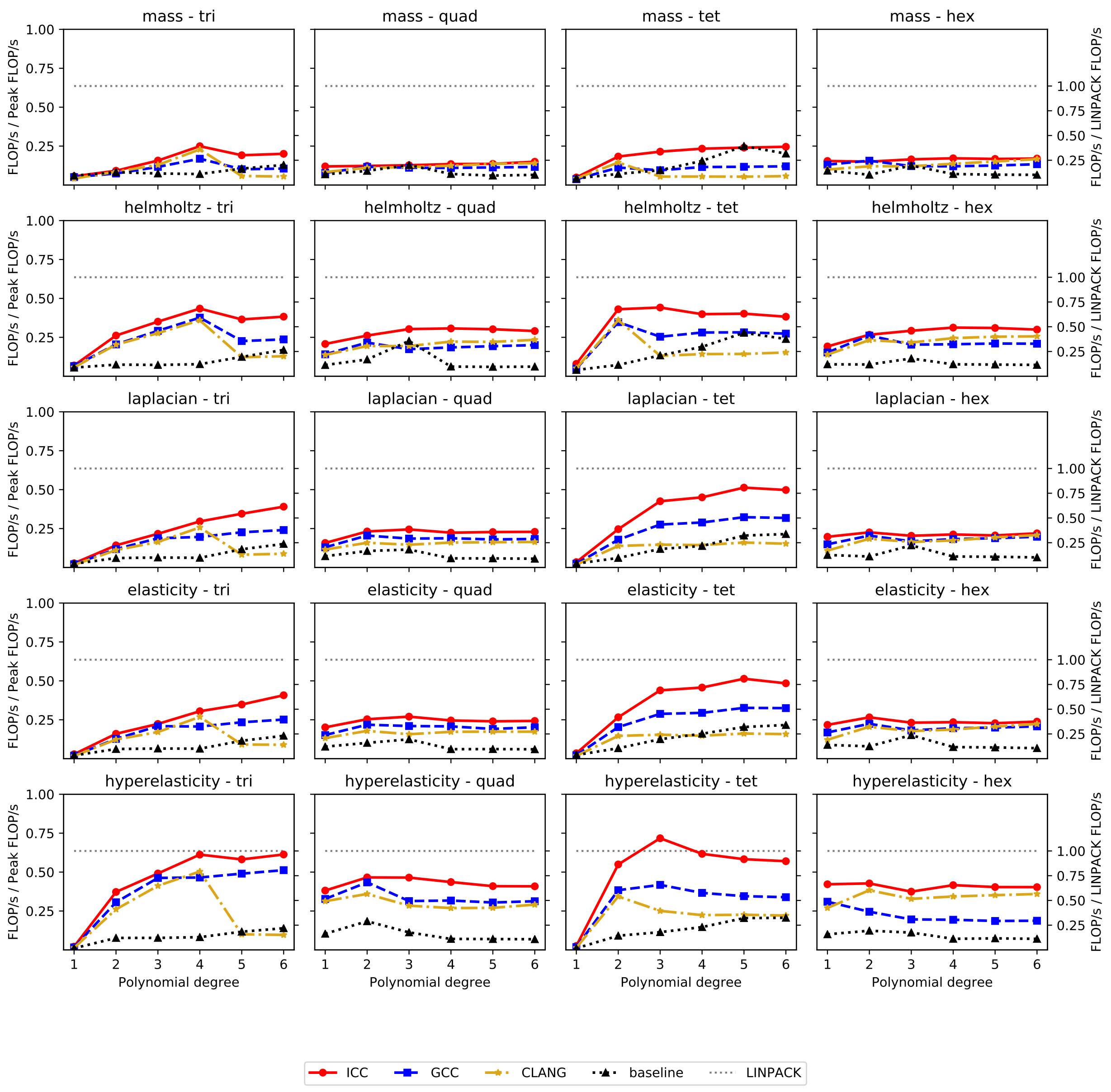

Figure 4.5: The fraction of peak FLOP/s (as listed in Table 4.1) achieved by different compilers for operators {`mass`, `helmholtz`, `laplacian`, `elasticity`, `hyperelasticity`}, on meshes {`tri`, `quad`, `tet`, `hex`} on **Skylake** using **OpenMP pragma** with 40 MPI processes. The dotted line indicates the fraction of peak performance achieved by the LINPACK benchmark.

.

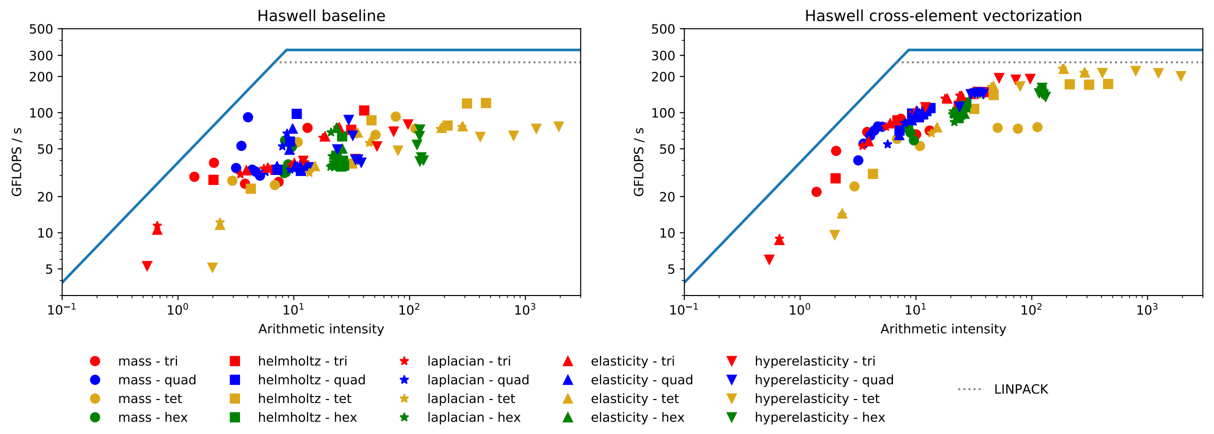

Figure 4.6: Roofline model of operators for basline and cross-element vectorisation using GCC on **Haswell**. The dotted lines indicate the performance of the LINPACK benchmark.

.

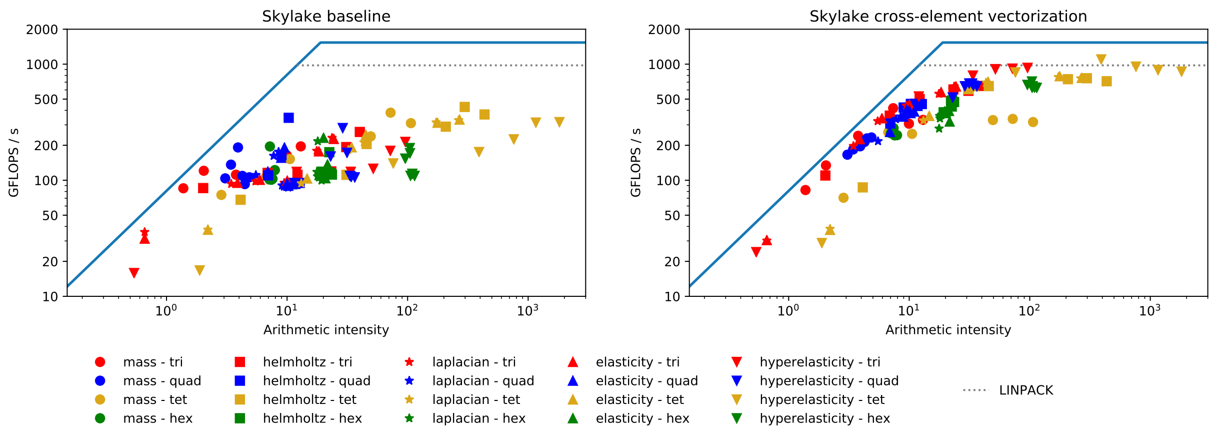

Figure 4.7: Roofline model of operators for basline and cross-element vectorisation using GCC on **Skylake**. The dotted lines indicate the performance of the LINPACK benchmark.

.

4.5.3 Compiler comparison and vector extensions

When vectorising with OpenMP SIMD directives, ICC gives the best performance for almost all test cases, followed by Clang, while GCC is significantly less competitive. The performance disparity is more pronounced on Skylake than on Haswell. However, when using vector extensions, Clang and GCC improve significantly and are able to match the performance of ICC on both Haswell and Skylake, whereas ICC performs similarly with both OpenMP SIMD directives and vector extensions.

We use the Intel Software Development Emulator to count the number of instructions executed at runtime for code generated by different compilers. The data, shown in Table 4.4, indicate that although floating-point operations are fully vectorised by all compilers when using OpenMP SIMD directives for cross-element vectorisation, GCC and Clang generate more load and store instructions between vector registers and memory. One possible reason is that GCC and Clang choose to allocate short arrays to the stack rather than the vector registers directly, causing more load on the memory subsystem.

Table 4.4: Statistics of memory instructions (in millions) executed by code generated by different compilers for degree 9 `helmholtz` kernel on `quad` mesh, running on Haswell.

Instruction type	**ICC**	**GCC**	**Clang**
memory read 32 bytes	1,346	1,448	1,621
memory write 32 bytes	631	1,000	824
Total	1,977	2,448	2,445

In light of these results, we conclude that vectorisation with vector extensions allows greater performance portability on different compilers and CPUs for our applications. It is, therefore, our preferred strategy for implementing cross-element vectorisation, and is the default option for the rest of our analysis in this chapter.

4.5.4 Vectorisation speed-up

Almost across the board, significant speed-up is achieved on the test cases under consideration. Slowdown occurs in two situations. First, on low polynomial degrees, the kernels tend to have low arithmetic intensity so that the increase in available floating-point throughput through cross-element vectorisation cannot compensate for the increase in the size of the working set of data. Second, on simple operators such as `mass` on `tri` and `tetra`, the kernels have simple loop structures and the compilers can sometimes successfully apply other optimisations such as unrolling and loop interchange to achieve vectorisation without batching elements in the outer loop. The pattern of speed-up is consistent across Haswell and Skylake. Higher speed-up is generally achieved on more complicated operators(e.g. `hyperelasticity`), and on tensor-product elements (`quad` and `hex`), which generally correspond to more complicated loop structure and higher arithmetic intensity due to the Jacobian recomputation at each quadrature point.

4.5.5 Achieved peak performance

We observe that the fraction of peak performance varies smoothly with polynomial degrees for cross-element vectorisation in all test cases. This fulfils an important design requirement for Firedrake: small changes in problem setup by the users should not create unexpected performance degradation. This is also shown in Figures 4.6 and 4.7 where the results are more clustered on the roofline plots after cross-element vectorisation. The baseline shows performance inconsistency, especially for low polynomial degrees. For instance, for the `helmholtz` operator with degree 3 on `quad`, the quadrature loops and the basis function loops all have trip counts of 4, which fits the vector length on Haswell and results in better performance.

On simplicial meshes (`tri` and `tetra`), higher order discretisation leads to kernels with very high arithmetic intensity because of the quadratic and cubic increases in the number of basis functions, and thus the loop trip counts. This is due to the current limitation that simplicial elements in Firedrake cannot be sum factorised, thus they do not reflect realistic use cases for higher order methods. In these test cases, we observe that the baseline approaches cross-element vectorisation for sufficiently high polynomial degrees. This is not a serious concern for our optimisation strategy because the break-even degrees are very high except for simple operators such as `mass`, and ultimately, tensor-product elements are more competitive for higher order methods in terms of algorithmic complexity. We note that further developments in TSFC and FInAT such as the Bernstein elements [Kirby and Thinh, 2012] would enable sum factorisation for simplicial meshes. These techniques might introduce loop-carried dependencies to the local assembly kernels, which could be challenging for intra-kernel vectorisation strategies but is composable with cross-element vectorisation, and the approach described here can still be applied.

We also observe that there exists a small number of test cases where the achieved peak performance is marginally higher than the LINPACK benchmark. Although not providing an explicit reason, McCalpin [2018] reported that the LINPACK benchmark could experience more than 10% slowdown on recent Xeon Platinum processors, potentially due to snoop filter evictions. Another possible reason for this observation is thermal throttling since our test cases typically run for a shorter period of time compared to LINPACK.

4.5.6 Tensor-product elements

We observe higher and more consistent speed-up for tensor-product elements (`quad` and `hex`) on both Haswell and Skylake. This is because, on these meshes, more computation is moved out of the innermost loop due to sum factorisation performed by TSFC, resulting in more challenging loop nests for the baseline strategy which attempts to vectorise within the element kernel. The same applies to the evaluation of the Jacobian of coordinate transformation, which is a nested loop over quadrature points after sum factorisation for tensor-product elements.

The base elements of `quad` and `hex` are interval elements in 1D, thus the extents of loops over degrees of freedom increase only linearly with respect to the polynomial degrees, as shown in Table 4.2. As a result, the baseline performance does not improve as quickly for higher polynomial degrees on `quad` and `hex` compared with `tri` and `tet`, resulting in stable speed-up for cross-element vectorisation observed on tensor-product elements.

4.6 Chapter summary

We have presented a portable, general-purpose solution for delivering stable vectorisation performance on modern CPUs for matrix-free finite element assembly. We have performed extensive experimental evaluations for a broad class of finite element operators on a large range of elements and polynomial degrees. We showed that compiler-based vector extensions facilitate generating appropriate vectorisable code, improving performance portability on all three C compilers that we tested. We described the implementation of cross-element vectorisation in Firedrake, which is transparent to the end-users. Although the technique of cross-element vectorisation is conceptually simple and has been applied in other works, our implementation based on code generation is automatic, robust and composable with other optimisation passes, such as the argument factorisation algorithm described in Chapter 3.

The newly introduced abstraction layer, together with the integration of Loopy in the code generation and optimisation pipeline, opens up multiple possibilities for future developments in Firedrake. These include code generation with intrinsic instructions, loop tiling, and GPU acceleration, which we will discuss in the next chapter.

Chapter 5

Global assembly of matrix-free operators on GPUs

Graphical Processing Units (GPUs) are engaging platforms for scientific computing that usually offer high (peak) performances as well as better cost and energy effectiveness. Systems empowered with GPUs frequently occupy the top spots in the list of the world's most powerful supercomputers[1]. However, despite the investments in these platforms and advances in the programming toolchains for GPUs, many simulations based on the finite element methods are still not ported to GPUs. In the previous chapter, we have demonstrated the integration of Loopy in Firedrake which enables cross-element vectorisation on CPUs. In this chapter, we describe how the representation of the global assembly process in Loopy allows us to generate CUDA and OpenCL code to target GPUs.

5.1 Motivation and related works

One particular challenge hindering more widespread usage of GPUs is the substantial effort required to develop GPU kernels or rewrite existing CPU code in order to leverage the architectural advantages of GPUs. Moreover, GPU implementations are often needed to be updated regularly to target devices from different vendors, or devices from different generations of the same vendor, as the GPU hardware architecture and instruction sets tend to be redesigned more frequently compared to CPUs.

As discussed in Section 2.1.4, using a matrix-free approach is another way to improve the performance of an iterative solver. This approach is even more applicable to GPU-based systems due to the advantages of hav-

[1]https://www.top500.org/

ing less memory transfer and better memory access patterns [Kiran et al., 2020]. GPU-based matrix-free implemenations have been developed for a wide range of applications in domains such as elasticity [Martinez-Frutos et al., 2015], heat conduction [Kiss et al., 2012], weather simulation [Müller et al., 2013, 2015], earthquake modelling [Komatitsch et al., 2009], fluid mechanics [Fehn et al., 2019] and topology optimisations [Ram and Sharma, 2017].

To implement the global assembly phase of a FEM solver using a matrix-free approach, it is possible to directly leverage the libraries shipped by the hardware vendors, such as cuBLAS and cuSPARSE, which provides certain basic linear algebra routines. This has been demonstrated by Müller et al. [2015] and Ljungkvist [2017]. However, significant effort is often needed to reorganise the computation in order to utilise these routines available in the libraries. This process tends to be application-specific and non-trivial for a general FEM framework like Firedrake. The performance of such approaches are usually not portable and highly dependent on problem sizes and discretisation schemes.

Alternatively, the computation kernels can be implemented at the CUDA or OpenCL level. Cecka et al. [2011] and Dziekonski et al. [2012] have manually ported the global assembly step from CPUs to GPUs for specific applications. This re-implementation is non-trivial and typically involves introducing a significant preprocessing stage to benefit from the fine-grain parallelism on GPUs. Compared to a library-based approach, such implementations are even more device-specific with limited reusability.

Other works such as [Bolz et al., 2003, Klöckner et al., 2009, Komatitsch et al., 2009] focus on specific applications and succeed in delivering high-performance GPU implementations. In particular, the particular PDEs and discretisation schemes used in these applications allow the authors to apply sophisticated data transformations to remove conflicts and promote data locality. While significant insights can be gained from these methods, it remains unclear how the techniques can be applied more generally and how much of the developed infrastructure in these works can be reused for future hardware.

All of the above-mentioned approaches involve manually implementing the computation kernels in CUDA or OpenCL. Pichler and Haase [2017] instead make use of THURST [Hoberock and Bell, 2010], a higher-level C++ templated library to allow faster prototyping. While it is likely easier to implement the solver in THURST than in CUDA, this work is still restricted to one application and one discretisation.

To generalise and automate the process of targeting GPUs, Markall et al. [2010] uses UFL [Alnæs et al., 2014] to transform the weak form of a PDE into a DAG of fine-grain computational operators, each of which is mapped to a pre-implemented CUDA kernel. This approach significantly raises the programming abstraction level for the users, who can specify the mathematical problems in UFL in many cases, but the need to manually implement computational kernels significantly limits the generalisability. We note that OP2 also supports GPU code generation for computations on unstructured mesh [Giles et al., 2013]. This functionality was partially ported to PyOP2 initially [Rathgeber et al., 2012] but has not been actively maintained and updated to keep up with the rest of the Firedrake components, missing key features such as the support for tensor product elements. Updating PyOP2 to match the functionality of OP2 is a significant endeavour that also requires maintaining a separate code generator in PyOP2 going forward.

The introduction of Loopy in the Firedrake framework, as described in Chapter 4, provides another pathway to automate global assembly on GPUs through code generation. This is because Loopy already support generating CUDA or OpenCL code from the same high-level descriptions of loops, such as the example in Listing 4.4. In this chapter, we describe how PyOP2 is modified to support GPUs. We extend PyOP2 to apply necessary transformations to the Loopy kernels, and leverage existing Loopy backends to generate the CUDA or OpenCL kernels that describe the global assembly computation. PyOP2 is also modified to generate the host-side code that organises data movement.

This approach completes the code generation pipeline in Firedrake on GPUs and allows users to target both CPUs and GPUs starting from the same problem definition, with the same flexibility in choosing different discretisation schemes. The loop transformation facilities in Loopy also enables easy exploration of different parallelisation strategies, where the loop nests are divided among GPU threads following different patterns.

The rest of this chapter is divided into two parts. In Section 5.2, we describe our methodology to target GPUs in Firedrake. We evaluate our approach with numerical experiments in Section 5.3.

5.2 Implementation

5.2.1 Implementation on CUDA and OpenCL

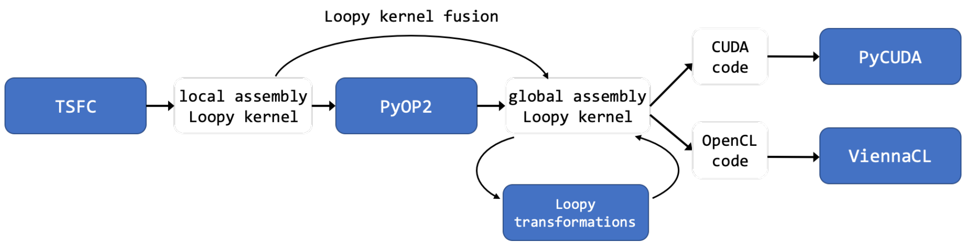

Figure 5.1: GPU backend implementation for CUDA and OpenCL devices in Firedrake using Loopy.

The task to target GPUs in Firedrake is two-fold. First, the device code (i.e. a kernel implemented in CUDA or OpenCL) for the global assembly computation needs to be generated. Second, the runtime system needs to be established to launch the kernels and orchestrate data movement between the host and the device.

Since Loopy natively supports CUDA and OpenCL as code generation targets, once a (backend-independent) global assembly Loopy kernel is created (such as the example in Listing 4.4), this same Loopy kernel can be used to generate not only CPU code, but also CUDA and OpenCL device kernels. Compared with generating code on CPUs, the CUDA and OpenCL kernels require different parallelisation strategies to match the iterations to the blocks and threads on GPUs, as explained later on.

CUDA

In Firedrake, many data structures in PyOP2 are essentially a thin layer of abstraction wrapped around corresponding PETSc objects. For example, a PyOP2 `Dat` that represents a discretised vector is an object owing a (distributed) array representing a PETSc `Vec`. When running on CPUs, PyOP2 directly operates on these arrays while executing the kernels. For GPU targets, PyOP2 needs to transfer the inputs to the device first (if they are not on the device already, or if they are on the device but are invalidated), launch the kernel and transfer the results back if they are needed on the host. The GPU runtime of Firedrake handles this process.

We use PyCUDA [Klöckner et al., 2012] as the runtime for the CUDA backend in Firedrake. PyCUDA provides a Python interface that allows easy integration with PyOP2 to compile and launch the CUDA kernels on GPUs. The memory allocation on the device and data movement between the host and the device are all handled through Python functions provided by PyCUDA. In this case, the integration with PETSc is "shallow": to offload a computation on GPU, PyOP2 obtains the pointers to the underlying memory of the PETSC `Vec`s, uses PyCUDA to send the data to the device (if needed) and launch the kernel with these pointers, There is, therefore, no need for explicit host-side code in this scenario.

For the OpenCL backend, we use ViennaCL [Rupp et al., 2016] for the runtime support. ViennaCL is a header-only library that provides a C++ interface for linear algebra routines. It is the recommended mechanism to leverage PETSc in OpenCL. Compared with CUDA, OpenCL provides a lower-level API, and the host-side programs are usually more elaborate. This observation also holds for us when integrating ViennaCL into PyOP2. Instead of handling the runtime in Python, we create a host-side C++ program using ViennaCL that compiles the kernel, transfers the data, sets computation properties such as work group sizes and launches the kernel. This program is non-trivial but nonetheless well-structured, and can be created at runtime from a template. A skeleton of such a host-side program is shown in Listing 5.1 where details such as error handling are omitted.

OpenCL

Listing 5.1: An example of host-side ViennaCL code to transfer data and launch OpenCL kernels with details such as error handling omitted.

```
#include <CL/cl.h>
#include "petsc.h"
#include "petscvec.h"
#include "petscviennacl.h"
#include "viennacl/ocl/backend.hpp"
#include "viennacl/vector.hpp"
#include "viennacl/backend/memory.hpp"

extern "C" void kernel_executor(cl_kernel kernel, int start, int end,
    Vec dat0, cl_mem map0, ...)
{
  // ViennaCL vector declarations.
  viennacl::vector<PetscScalar> *dat0_viennacl;
  // ... More declarations ... //

  // Get ViennaCL Vec from PETSC Vec.
  VecViennaCLGetArray(dat0, &dat0_viennacl);
  // ... More similar instructions ... //

  // Set the kernel arguments.
  viennacl::ocl::handle<cl_mem> dat0_handle = ...;
  clSetKernelArg(kernel, 0, sizeof(PetscScalar), &dat0_handle);
  // ... More similar instructions ... //

  // Get the context.
  viennacl::ocl::context ctx = ...;

  // Get the command queue.
  cl_command_queue queue= ctx.get_queue().handle().get();

  // Set the local and global work group sizes.
  const int dim = ...;
  size_t l_size[dim] = { ... };
  size_t g_size[dim] = { ... };

  clFinish(queue);

  // Enqueue the kernel.
  clEnqueueNDRangeKernel(queue, kernel, dim, g_size, l_size);

  clFinish(queue);

  // Restoring the arrays to the petsc vecs
  VecViennaCLRestoreArray(dat0, &dat0_viennacl);
  // ... More similar instructions ... //
}
```

We also note that the PETSc data objects for GPU targets need to be initialised with the corresponding PETSc types, so that the data formats and layout semantics match what is expected by the PETSc algorithms.

This means, for example, a PETSc `Vec` (owned by a PyOp2 `Dat`) should be created with the type `VECCUDA` (CUDA) or `VECVIENNACL` (OpenCL).

The exceptions to the above are PyOP2 `Maps`, which are used to represent the mapping from an entry in a source set (e.g. a cell in a mesh) to one or more entries in a target set (e.g. one or more DOFs in a finite element function space). In PyOp2, `Maps` are represented by multi-dimensional Numpy arrays of integers, and, unlike `Dats`, there are no underlying PETSc objects for `Maps`. To transfer the (read-only) `Maps` to the device before launching a kernel, on the CUDA backend, we copy the array to the device using PyCUDA functions. In the case of OpenCL, we generate and execute a host-side ViennaCL program such as the example in Listing 5.2 to perform this copy.

Listing 5.2: A helper host-side ViennaCL program to copy the data representing a PyOP2 `Map` to the device.

```
#include <CL/cl.h>
#include "petsc.h"
#include "petscviennacl.h"

extern "C" cl_mem get_map_buffer(int * __restrict__ array, const int
    array_size, viennacl::ocl::context ctx)
{
  int size = sizeof(cl_int) * array_size;
  return clCreateBuffer(ctx.handle().get(),
    CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, size, array);
}
```

In general, although the OpenCL and CUDA backends are functionally similar, the support for CUDA in our toolchain is more established. It is also easier to use as it operates at a higher level and is better integrated with Python. We note that there also exists a Python binding for ViennaCL, which might remove the need for generating the host-side code. This path has not been sufficiently explored, partly due to the fact that it abstracts away the runtime aspects completely and makes it harder to use this binding in PyOP2 which is responsible for organising data movements. As a result, we only describe the implementation and performance details for the CUDA backend and use CUDA nomenclature and conventions in the rest of this chapter.

5.2.2 Atomic addition

Listing 5.3: CUDA code snippet that implements atomic incremems for double-precision floating-point numbers.

```
#if __CUDA_ARCH__ < 600
__device__ double atomicAdd(double *address, double val) {
  unsigned long long int *address_as_ull =
    (unsigned long long int *)address;
  unsigned long long int old = *address_as_ull, assumed;

  do {
    assumed = old;
    old = atomicCAS(address_as_ull, assumed,
      __double_as_longlong(val + __longlong_as_double(assumed)));

  } while (assumed != old);

  return __longlong_as_double(old);
}
#endif
```

In the global assembly process on GPUs, typically, the results are stored in the device global memory during the computation. If different threads could compute the contributions for the same DOFs, the updates to the global data need to be implemented as an atomic update to mitigate potential race conditions. However, on certain Nvidia GPUs (for example, the devices from the GeForce product line before the Pascal series), `atomicAdd` function is not implemented for double-precision floating-point numbers in the CUDA API. Instead, we extend the CUDA code generator in Loopy to generate the code snippet in Listing 5.3, which implements the atomic addition with a 64-bit atomic *compare and swap* (`CAS`) instruction on such devices.

5.2.3 Single element per thread parallelisation

As illustrated in Algorithm 4, there are multiple loops in the global assembly routine that can be parallelised. These include:

- The outer loop over elements in the domain
- The reduction loop over basis functions
- The reduction loop over quadrature points

The most straightforward strategy to parallelise the global assembly loop nests on GPUs is to assign each iteration of the outer loop to one thread so that each thread computes the contribution from one element in the domain. The computation on each cell can be performed independently, and no synchronisation is required between threads during the computation. The write-back to the global vector needs to be performed by atomic additions as shown in Section 5.2.2, due to potential race conditions. This strategy is analogous to the cross-element vectorisation strategy on CPUs as described in Chapter 4, but treating one GPU thread as a vector line on CPU.

A critical feature of the programming abstraction on GPUs is the concept of hierarchical parallelism, where threads are grouped into thread blocks. In the hardware view, threads in the same blocks run on the same stream processor. They can communicate with each other via shared memory and synchronise with block barriers. This means in addition to establishing the

Listing 5.4: The global assembly Loopy kernel of the Helmholtz operator (degree 2, on triangles) transformed for generating CUDA code.

```
KERNEL: helmholtz_gpu_sept
---------------------------------------------------------------------
ARGUMENTS:
start: int32
end: int32
dat0: type: float64, space: global
// ... More arguments ... //
---------------------------------------------------------------------
DOMAINS:
[end, start] -> { [n_outer, n_inner] :
  n_inner >= start - 32n_outer and
  0 <= n_inner <= 31 and n_inner < end - 32n_outer }
// ... More temporaries ... //
---------------------------------------------------------------------
INAME_IMPLEMENTATION_TAGS:
n_outer: g.0
n_inner: l.0
---------------------------------------------------------------------
TEMPORARIES:
t0: type: float64, shape: (1, 6), space:auto
form_t14: type: float64, shape: (6, 6), space:global
// ... More temporaries ... //
---------------------------------------------------------------------
INSTRUCTIONS:
for n_outer, n_inner
  for i6
    t2[0, i6] = dat2[map0[n_inner + n_outer*32, i6]]
  end i6
  // ... More instructions ... //
  for i8
    dat0[map0[n_inner + n_outer*32, i8]] += t0[0, i8]  {atomic=update}
end n_outer, i8, n_inner
```

task for a thread, we also need to specify how many threads comprise a block.

Using the single-element-per-thread parallelisation strategy, because each thread can run independently and update the global memory directly, there is no need for communication between threads and an arbitrary block size can be chosen. In this work, we have picked the block size of 32 because it matches the warp size on the hardware size. We also verified that as long as the chosen block size is a multiple of 32 and less than the hardware block size limit of 1024, the differences in performance are insignificant for this parallelisation strategy.

Listing 5.4 shows how the Loopy kernel for global assembly of the Helmholtz operator in Listing 4.4 are transformed to generate CUDA code. We split the outer loop `n` over the cells into `n_outer` and `n_inner`, with the extent of `n_inner` equals to 32. `n_outer` is assigned the tag `g.0`, which maps it to blocks, while `n_inner` is assigned the tag `l.0`, which maps it to individual threads. This means they are mapped to the built-in variables `blockIdx.x` and `threadIdx.x` in CUDA.

Listing 5.5: CUDA kernel for the action of the Helmholtz operator (degree 2, on triangles).

```
// ... Definition of atomicAdd in Listing 5.3 ... //
__constant__ double const form_t13[6] = { ... };
__constant__ double const form_t14[6 * 6] = { ... };
__constant__ double const form_t15[6 * 6] = { ... };
__constant__ double const form_t16[6 * 6] = { ... };

extern "C" __global__ void __launch_bounds__(32) helmholtz(
  int const start, int const end, double *__restrict__ dat0,
  double const *__restrict__ dat1, double const *__restrict__ dat2,
  int const *__restrict__ map0, int const *__restrict__ map1)
{
  if (32*(blockIdx.x+start-(31+31*start)/32)+threadIdx.x-start >= 0 &&
      end-32*(blockIdx.x+start-(31+31*start)/32)-threadIdx.x > 0)
{
  double form_t0;
  // ... Temporary variable declarations ... //

  for (int i6 = 0; i6 <= 5; ++i6)
    t2[i6] = dat2[map0[6 * (32 * blockIdx.x + threadIdx.x) + i6]];
  for (int i2 = 0; i2 <= 2; ++i2)
    for (int i3 = 0; i3 <= 1; ++i3)
      t1[2 * i2 + i3] =
        dat1[2 * map1[3 * (32 * blockIdx.x + threadIdx.x) + i2] + i3];
  for (int i0 = 0; i0 <= 5; ++i0)
    t0[i0] = 0.0;
  form_t0 = -1.0 * t1[0];
  // ... More similiar instructions ... //
  for (int form_ip = 0; form_ip <= 5; ++form_ip)
  {
    form_t17 = 0.0;
    // ... More similiar instructions ... //
    for (int form_i = 0; form_i <= 5; ++form_i)
    {
      form_t17 += form_t16[6 * form_ip + form_i] * t2[form_i];
      // ... More similiar instructions ... //
36      }
37      // ... More similiar instructions ... //
38      for (int form_j = 0; form_j <= 5; ++form_j)
39        t0[form_j] +=
40          form_t15[6 * form_ip + form_j] * form_t23 +
41          form_t16[6 * form_ip + form_j] * form_t25 +
42          form_t14[6 * form_ip + form_j] * form_t24;
43    }
44    for (int i8 = 0; i8 <= 5; ++i8)
45      atomicAdd(&dat0[map0[6 * (32 * blockIdx.x + threadIdx.x) + i8]],
46                t0[i8]);
47  }
48  }
```

The generated CUDA kernel is shown in Listing 5.5. We note Loopy correctly identifies the write back to the global memory in lines 45-46 to be an atomic update, and an `atomicAdd` instruction is generated as a result.

The NVCC compiler does not reliably unroll small loops. Instead, we attach the inames that correspond to the (innermost) loop over the physical dimensions with the implementation tag `unr` so that Loopy always unrolled them for performance consistency. Note that such loops might not exist in the kernel if all the functions in the equation are scalar functions.

5.2.4 Global constants

In a Loopy kernel, temporary variables (i.e. data that are not arguments of the kernel) can be assigned a memory *address space*, as shown in Listing 5.4. The address space determines the storage allocation of the temporary variable. Possible spaces include: `PRIVATE` (thread-specific memory), `LOCAL` (block-specific memory) and `GLOBAL` (global memory). It is usually sufficient to set the address space to `AUTO` which allows Loopy to determine the memory location automatically. Nonetheless, we assign the `GLOBAL` space to the constant arrays that describe the tabulations of the basis functions and quadrature weights. As shown in Listing 5.5, these arrays are declared with `__constant__` in the generated CUDA kernel and are allocated in the read-only constant memory on the device.

However, the size of the constant memory is limited to 64KB in recent NVidia GPUs[2]. On the other hand, the dimension of the tabulation matrices is $N_{quadrature} \times N_{basis}$, which means the constant memory is not large enough for these constants of complicated equations on higher orders. This problem is more pronounced on simplicial meshes (`tri` and `tetra`) because the numbers of quadrature points and basis functions grow quadratically or cubically (due to the lack of sum-factorisation by the form compiler), as shown in Table 4.2.

To mitigate this problem, we design a transformation that copies these constant arrays to the global memory on the device and pass them to the

[2]https://docs.nvidia.com/cuda/cuda-c-programming-guide

Table 5.1: Hardware specification for experiments

	GeForce GTX TITAN	**Tesla P100**
Microarchitecture	Kepler	Pascal
Base frequency	837 MHz	1126 MHz
CUDA cores	2688	3584
FMA instructions available	Yes	Yes
Double-precision cost factor	3	2
Peak performance (double-precision)[3]	1499.9 GFLOP/s	4035.6 GFLOP/s
LINPACK performance (double-precision)[4]	768.7 GFLOP/s	3967.0 GFLOP/s
Device memory	6 GB GDDR5	16 GB HBM2
Device memory bandwidth[5]	241.4 GB/s	500.2 GB/s
Block size	32	32

kernel as additional arguments. This transformation is achieved by removing these arrays from the list of temporary variables in the Loopy kernel and adding them to the list of kernel arguments, with the computation remaining unchanged. This strategy allows Firedrake to support all such assembly kernels on GPU, at the cost of requiring additional loads from the global memory.

5.3 Performance evaluation

In this section, we evaluate the performance of the global assembly kernels of a suite of matrix-free operators on GPUs.

5.3.1 Experimental setup

We performed experiments on two systems with NVidia GPUs from the GeForce (desktop-orientated) and Tesla (server-orientated) product lines. The details are listed in Table 5.1. We use CUDA toolkit version 9.2 for this experiment.

Because many GPUs are optimised for 32-bit arithmetics, they can be ineffective to perform 64-bit computations. This situation is particularly relevant for devices from the GeForce product line which are optimised for desktop graphics and gaming uses. This inefficiency is noted as *double-precision cost factor* in Table 5.1, where a cost factor of 2 indicates there is no loss of efficiency compared with executing single-precision computations.

[4]Calculated as: *base frequency* $\times$ *#cores* $\times$ 2 *(for FMA)* $\div$ *double-precision cost factor*

[5]NVidia LINPACK Benchmark. *https://developer.nvidia.com/rdp/assets/cuda-accelerated-linpack-linux64*

We use the *Base Frequency* provided by the NVidia to calculate the peak performance in Table 5.1. Similar to Chapter 4, we run the LINPACK benchmark provided by the vendor to obtain estimates for achievable peak performance.

We perform the evaluation using the same family of operators and discretisations listed in Table 4.2. The computations are repeated between 100 to 1000 times.

On most NVidia GPUs, the L1 cache of a streaming processor is backed by the same memory as the shared memory. Because the kernels do not use shared memory in the single-cell-per-thread parallelisation strategy, we set the cache configuration to `PreferL1` in the CUDA runtime in PyOP2, so that more space is allocated to the L1 cache.

5.3.2 Experimental results and discussion

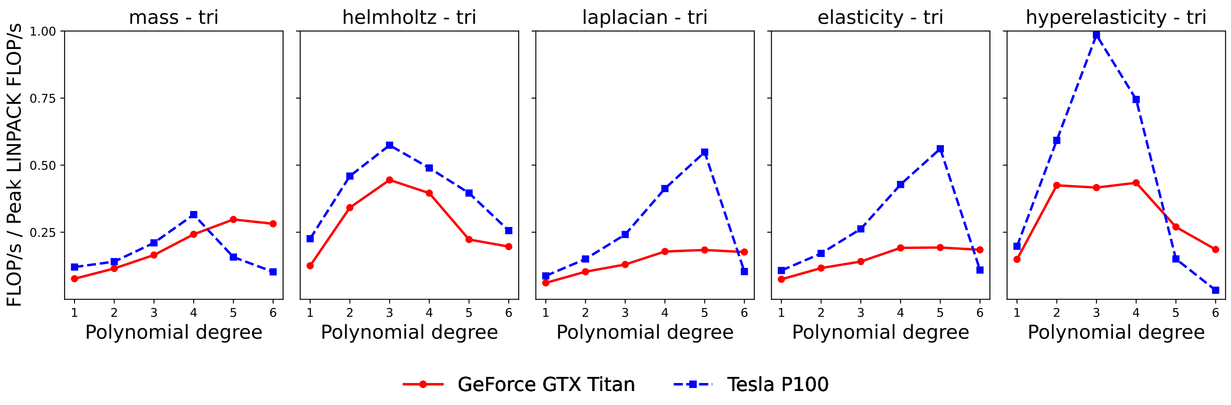

Figure 5.2: The fraction of peak LINPACK FLOP/s (as listed in Table 5.1) achieved for operators {`mass, helmholtz, laplacian, elasticity, hyperelasticity`}, on mesh {`tri`} on **GeForce GTX Titan** and **Tesla P100**. Parallelised using the single-element-per-thread strategy with block size of 32.

Figure 5.2 shows the performance of the operators on triangular meshes on the two GPU devices. We observe that the performance improves with increasing polynomial degrees at low polynomial degrees. This observation corresponds to the increase in the arithmetic intensities, as shown in Table 4.2, that improves the utilisation of the devices. This is also evident in Figures 5.3 and 5.4 which show the roofline models of the operators on the two devices, respectively.

However, at higher polynomial degrees, the performance starts to decrease with increasing degrees. One important reason for this observation is the higher register usage of the generated CUDA kernels at higher polynomial degrees. On GPUs, each streaming process owns a register file shared

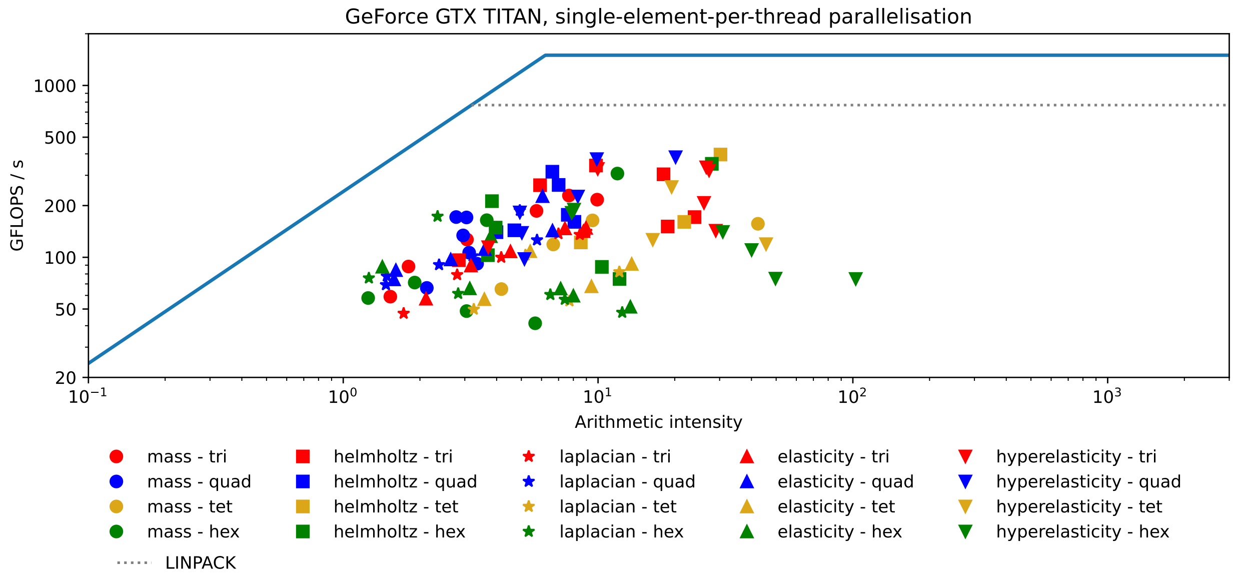

Figure 5.3: Roofline model of operators for single-element-per-thread parallelisation on **GeForce GTX TITAN**. The dotted lines indicate the performance of the LINPACK benchmark.

.

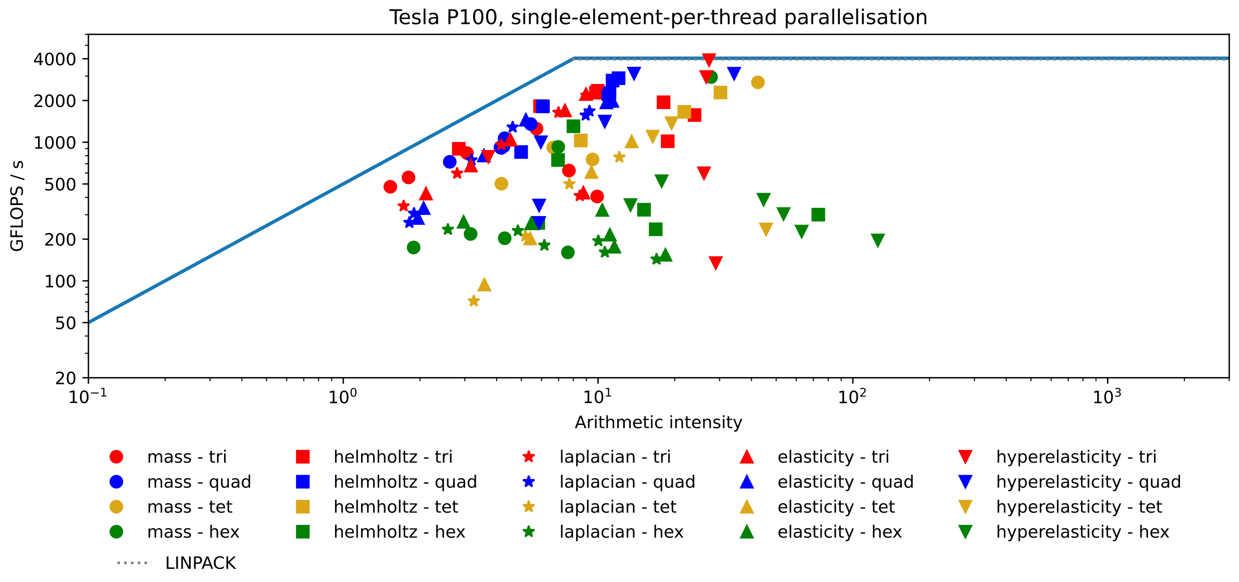

Figure 5.4: Roofline model of operators for single-element-per-thread parallelisation on **Tesla P100**. The dotted lines indicate the performance of the LINPACK benchmark.

.

among all (active) threads. As a result, a higher number of registers allocated per thread will lead to lower thread occupancy and lower performances. In addition, register spilling increases memory traffic and instruction counts, further decreasing performances. This phenomenon is more evident for complicated operators such as `hyperelasticity`. Table 5.2 lists the register usage of the kernels as reported by the CUDA NVCC compiler. Nonetheless, high performance is achievable in some instances where the arithmetic intensity is high enough while register spilling does not significantly impact the performance yet, for example, the `hyperelasticity` operator on `tet` at degree 3.

Tensor-product elements

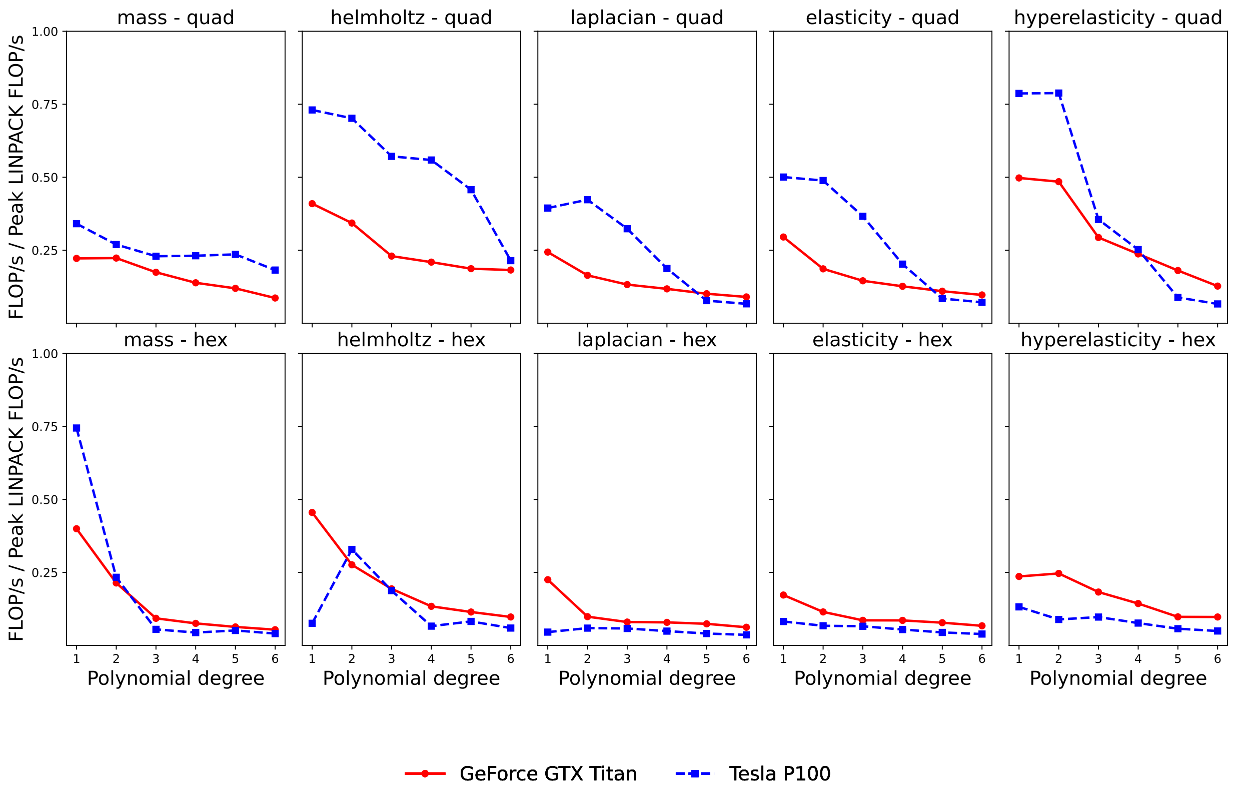

Figure 5.5: The fraction of peak LINPACK FLOP/s (as listed in Table 5.1) achieved for operators {`mass, helmholtz, laplacian, elasticity, hyperelasticity`}, on meshes {`quad, hex`} on **GeForce GTX Titan** and **Tesla P100**. Parallelised using the single-element-per-thread strategy with block size of 32.

Figure 5.5 shows the performance of the operators using tensor-product discretisations, namely, on `quad` and `hex` meshes. We observe that, at lower

Table 5.2: Register statistics of the generated CUDA kernels using single-element-per-thread parallelisation. R: register used. L: register spill loads (bytes). S: register spill stores (bytes). X: compilation failure (due to kernel using too much global constant memory).

		`tri`						`quad`						`tet`						`hex`					
		Titan			P100			Titan			P100			Titan			P100			Titan			P100		
	P	R	L	S	R	L	S	R	L	S	R	L	S	R	L	S	R	L	S	R	L	S	R	L	S
mass	1	26	0	0	32	0	0	44	0	0	50	0	0	48	0	0	44	0	0	122	0	0	110	0	0
	2	50	0	0	64	0	0	76	0	0	64	0	0	72	0	0	64	0	0	255	148	148	255	256	256
	3	68	0	0	64	0	0	108	0	0	112	0	0	128	0	0	128	0	0	255	992	980	255	1520	1496
	4	128	0	0	126	0	0	128	0	0	168	0	0	255	0	0	255	140	140	168	3796	3420	168	3904	3856
	5	128	0	0	128	0	0	168	0	0	255	0	0	X	X	X	X	X	X	255	1952	1956	255	1904	1904
	6	168	0	0	254	0	0	255	308	308	255	496	496	X	X	X	X	X	X	255	640	640	255	520	528
helmholtz	1	40	0	0	48	0	0	78	0	0	78	0	0	64	0	0	64	0	0	228	0	0	236	0	0
	2	74	0	0	64	0	0	136	0	0	144	0	0	128	0	0	128	0	0	255	608	608	255	620	620
	3	126	0	0	96	0	0	192	0	0	212	0	0	255	92	92	254	0	0	255	1872	1364	255	2380	1868
	4	128	0	0	168	0	0	244	0	0	255	52	52	X	X	X	X	X	X	255	4156	3588	255	4040	3416
	5	248	0	0	254	0	0	255	240	284	255	384	380	X	X	X	X	X	X	255	1888	1512	255	1816	1456
	6	255	180	176	254	0	0	255	564	620	255	760	756	X	X	X	X	X	X	255	2520	1880	168	3764	3516
laplacian	1	42	0	0	44	0	0	92	0	0	96	8	8	76	0	0	78	0	0	255	436	436	255	500	500
	2	66	0	0	64	0	0	212	0	0	232	0	0	168	0	0	168	0	0	168	4396	4292	168	4004	3944
	3	108	0	0	128	0	0	255	192	212	255	304	300	255	900	896	255	1448	1268	168	5332	5080	168	4720	4488
	4	168	0	0	254	0	0	255	632	712	255	824	884	168	4496	4620	64	6344	7072	255	4176	3456	255	4088	3488
	5	255	156	156	255	36	36	168	2184	2276	168	2496	2524	X	X	X	X	X	X	255	6440	5872	255	6240	5656
	6	255	516	516	255	720	720	168	3544	3548	168	3332	3264	X	X	X	X	X	X	255	3528	3480	255	3496	3504
elasticity	1	46	0	0	44	0	0	98	0	0	92	0	0	76	0	0	78	0	0	255	312	312	255	308	308
	2	66	0	0	64	0	0	224	0	0	238	0	0	168	0	0	228	0	0	168	4460	4356	168	4020	3968
	3	110	0	0	128	0	0	255	192	212	255	296	292	255	924	920	255	1608	1300	168	5356	5104	168	4772	4540
	4	168	0	0	254	0	0	255	624	696	255	824	884	168	4352	4212	64	5656	5704	255	4176	3464	255	4088	3488
	5	255	180	180	255	76	76	168	2184	2276	168	2496	2524	X	X	X	X	X	X	255	6496	5920	255	6272	5680
	6	255	516	516	255	720	720	168	3544	3548	168	3332	3264	X	X	X	X	X	X	255	3520	3480	255	3504	3512
hyperelasticity	1	70	0	0	64	0	0	204	0	0	218	0	0	128	0	0	126	0	0	255	932	908	255	880	796
	2	128	0	0	168	0	0	255	168	204	255	240	236	255	228	288	255	512	316	255	5716	5288	168	6332	6948
	3	228	0	0	239	0	0	255	592	592	255	640	640	255	1684	1928	255	2408	2364	255	3816	3288	255	3832	3448
	4	255	164	164	255	268	268	168	2248	2252	255	1484	1292	X	X	X	X	X	X	255	6032	5312	255	6400	6024
	5	255	548	548	255	708	708	168	3680	3684	168	3984	3868	X	X	X	X	X	X	168	8196	7916	255	5704	5168
	6	255	1052	1052	64	3660	3688	168	5552	5556	168	6760	6980	X	X	X	X	X	X	255	1792	1576	255	1808	1604

polynomial degrees, assembly on tensor-product elements tends to perform better compared with simplicial finite elements. This is because the kernels for tensor-product elements usually have higher arithmetic intensities at low polynomial degrees. However, similarly to the simplicial cases, the performance degrades at higher polynomial degrees due to register pressure. We also note that the arithmetic intensities do not increase with the polynomial degree for tensor-product elements, thus the performance almost uniformly decreases with increasing polynomial degrees.

Global constant memory

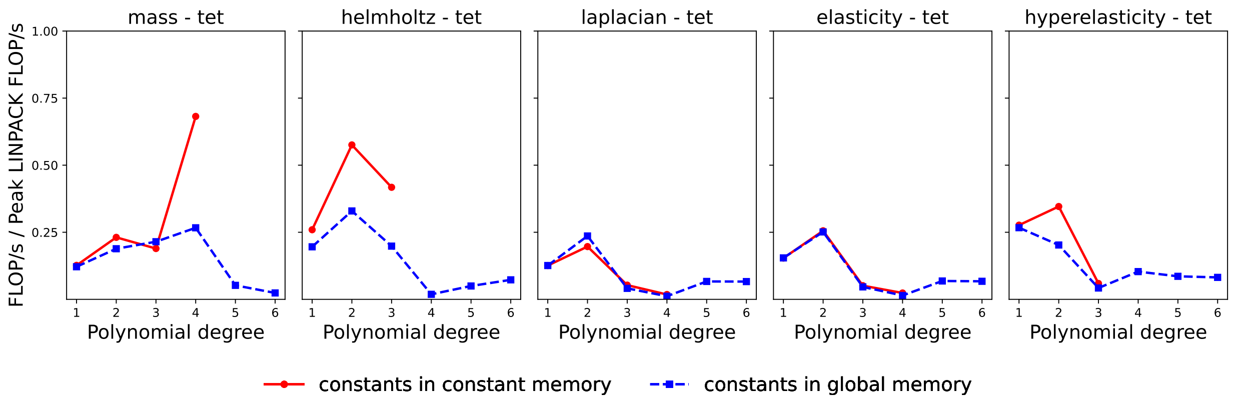

Figure 5.6: The fraction of peak LINPACK FLOP/s (as listed in Table 5.1) achieved for operators {`mass, helmholtz, laplacian, elasticity, hyperelasticity`}, on mesh {`tet`} on **Tesla P100**. Parallelised using the single-element-per-thread strategy with block size of 32.

As described in Section 5.2.4, the constant arrays that represent the tabulation and quadrature weights are stored in the global constant memory on the device, and accessed by all threads during computation. However, at higher polynomial degrees, the sizes of these arrays might exceed the capacity of the constant memory, results in compilation failures. Such cases are marked as **X** in Table 5.2. One plausible strategy to mitigate this issue, in general, is to store these constants in the global memory on the device and pass them into the kernels as additional arguments.

Figure 5.6 describes the effect of transferring the constants as kernel arguments for the `hyperelasticity` kernel on Tesla P100. In cases where the constant memory is large enough to accommodate the constant arrays, transferring them as additional arguments introduces more bandwidth pressure, resulting in lower performances typically. However, this strategy is successful in compiling this complicated kernel at high polynomial degrees. It provides a valuable fallback mechanism to a general framework such as

Firedrake, which needs to support arbitrary operators and discretisations specified by the users.

5.3.3 Limitations

The work presented in this chapter can be regarded as a proof of concept to add GPU support in Firedrake. There are many possible alternatives and extensions that have not been thoroughly explored. Apart from mapping each iteration of the outer loop over mesh entities to one thread, there are numerous strategies to partition the loop nests in a global assembly kernel, such as the thread transposition algorithm by Knepley et al. [2016] that makes use of the on-device shared memory to divide the computation for one entity among a group of threads, achieving better device utilisation in some instances. Although the Loopy abstraction can facilitate the kernel transformations required to implement such strategies, the difficulty lies in choosing a particular strategy and deciding its parameters (such as thread block sizes) to maximise performance. This is likely to require augmenting PyOP2 with a robust cost model and/or an auto-tuning setup (such as the one implemented in OP2 [Giles et al., 2013]) to guide the optimisation decisions.

Other than using atomic instructions, there are other strategies to mitigate potential data race hazards due to indirect accesses. These include using memory barriers to guard the data updates, and scheduling the instructions (e.g. using a graph colouring algorithm [Ljungkvist, 2017]) to avoid accessing the same data element from different threads.

Chapter 6

Summary and outlook

In this thesis, we have studied the design of intermediate representations of Firedrake and described certain adaptations that improve its code generation and optimisation capability. We summarise our findings in this final chapter and discuss possible future research directions.

In Chapter 3, we argue that GEM, a language based on symbolic tensor algebra, concisely and conveniently describes the local assembly phase of a finite element solver. Local assembly kernels can be represented as tensor contractions and structured as DAGs in GEM, where optimisations are implemented as rewriting of the DAGs following mathematical rules.

We have demonstrated the effectiveness of this approach by lifting the loop-invariant code motion optimisation, previously implemented in COFFEE as AST manipulations, to this higher abstraction layer of tensor algebra. Our experiments confirm that this particular optimisation is efficient and consistently outperforms the status-quo in terms of compilation speed and optimisation effectiveness.

However, there are other types of optimisations that can be and should be lifted to the level of symbolic tensor algebra. For example, certain kernels can be refactored such that the subexpressions involving compile-time constant tensors are organised together and *precomputed* in the generated code. Precomputation is not strictly beneficial as it could lead to a larger memory footprint. In addition, it might impact the effectiveness of other optimisations, e.g. by removing certain loop-invariant subexpressions which could otherwise be hoisted out, and thus should not be applied unconditionally in isolation.

More broadly, the sequence of applying a set of transformations to a DAG representing a local assembly kernel could impact the optimisation result, and TSFC would need to be enhanced with more sophisticated performance modelling in the future to determine such a sequence for a given kernel. An alternative strategy is to design new optimisation modes based on heuristics and pick the best result for a particular application after trying all modes. Because DAG analysis and refactorisation of symbolic expressions can be performed efficiently in the GEM representation, TSFC can, in theory, explore a large combination of various transformations without the compilation time becoming the bottleneck in Firedrake.

In Chapter 4, we have presented a portable, general-purpose solution for delivering stable vectorisation performance on modern CPUs for matrix-free finite element assembly. We have performed extensive experiments on a broad class of finite element operators on a large range of discretisation schemes. Although the technique of cross-element vectorisation is conceptually simple and has been applied in hand-written kernels before, our implementation based on code generation is automatic, transparent to the end-users, and composable with other optimisation passes in Firedrake.

In the experimental evaluation, we have presented results on two recent Xeon processors and compared the vectorisation performances of three popular C compilers to verify the performance portability of our approach. One important finding is that compiler-based vector extensions are productive mechanisms to obtain consistent vectorisation results across all three compilers.

One shortcoming of our algorithm is that the write-back to the global data structure is not vectorised due to possible race conditions. The newly introduced Conflict Detection instructions in the Intel AVX512 instruction set could potentially mitigate this limitation [Zhang, 2016, Section 2.3]. As a development in the future, new tags could be added to Loopy in order to support code generation targeting these processor intrinsics.

So far, we have focused on the matrix-free finite element method because it is compute-intensive and is therefore more likely to benefit from vectorisation. However, our methodologies and implementation can support matrix assembly with minimal modifications – a global assembly kernel needs to read from and write to entries of a global (sparse) matrix instead of a global vector. Firedrake relies on PETSc [Balay et al., 2017] to handle distributed matrices. When updating entries of a global matrix, PETSc requires specific data layouts for the input array. In the cases when several elements are batched together for cross-element vectorisation, PyOP2 needs to generate code to explicitly unpack and transpose the local assembly results into individual arrays before calling PETSc functions to update the global sparse matrices for each element. Future improvement could include eliminating this overhead, possibly by extending the PETSc API.

In Chapter 5, we describe our approach to adding GPU support in Firedrake by leveraging the code generation facility of the newly introduced Loopy abstraction. To demonstrate the generalisability of our strategy, we have conducted experimental evaluations on two GPU devices, using a test suite of a large family of operators and discretisations of various polynomial degrees.

Compared with CPUs, certain hardware or ISA features (such as hardware prefetchers and branch predictors) are less sophisticated on GPUs. The memory hierarchy is also less lenient in handling problematic access patterns and register spillings. Furthermore, GPU kernel compilers such as NVCC are less capable in performing aggressive optimisations automatically. As a result, our single-element-per-thread parallelisation strategy, although able to support a wide range of applications in Firedrake, cannot deliver predictable and portable performance on GPUs in general. More refined and targeted transformations are likely needed in the future. One potential future development is to partition the computation for each mesh entity among different threads, which synchronise via the on-device shared memory. Such parallelisation strategies are likely to reduce the register pressure which we identify as one of the performance bottlenecks. The difficulty in adding these strategies to PyOP2 is that they are likely to require kernel-specific fine-tuning for parameters such as block sizes and synchronisation patterns, making generalisability a challenging task for an application-agnostic tool like Firedrake. For example, the thread transposition algorithm by Knepley et al. [2016] only performs well in its original form if the least common multiple of N_q (the number of quadrature points) and N_b (the number of basis functions) is relatively small. In addition, this optimisation step could be an iterative process where hyper-parameters of the strategy are determined incrementally based on the performance statistics returned by the kernel compiler. In that consideration, the ability of Loopy to perform and compose kernel transformations efficiently will be essential in implementing such general parallelisation strategies, so that multiple strategies can be explored without jeopardising the compilation speed to a prohibitive extent.

Computationally, one important difference of the global assembly phase compared with the local assembly phase is accessing global data structure through indirect mappings. In local assembly kernels, GEM predominately only deals with dense linear algebra. In global assembly kernels, indirect accesses like `B[A[i]]` are analysed in Loopy by tools based on the polyhedral model. In the future, the GEM language could be extended to support limited cases of using expressions like `A[i]` as indices, so that optimisations like cross-element vectorisation can also be lifted to the higher level of symbolic tensor algebra, blurring the boundary between local and global assembly kernels. Certain domain-specific languages and compilers, such as Tensor Comprehension [Vasilache et al., 2018] in the domain of artificial neural network, already successfully support such access patterns. On a cautionary note, a valuable asset of GEM is that GEM is a simple language with very few constructs and concepts, and the drawbacks of extending GEM in this way include a looser GEM grammar and weakened invariant properties of the algebra, which could impact the ability of the GEM compiler to analyse and transform the expressions, potentially making DAG rewrites more complicated during optimisation.

More broadly, the software stack in Firedrake today is organised as multiple abstraction layers that are isolated from each other. Although this design helps to maintain invariant properties of each abstraction, we also note that there could be significant benefits in allowing intermediate representations of different abstraction levels to coexist. The recently announced MLIR project [Lattner et al., 2020] facilitates building intermediate representations on the same infrastructure, and thereby promoting a much more fine-grained style in instruction lowering and optimisation. As semi-conductor technology continues to advance, new hardware designs are constantly emerging to handle computation-intensive applications. With many of the recent processors, such as the Graphcore IPU, the Cerebras Wafer Scale Engine and the SymbaNova Dataflow System, deviate considerably from the traditional von Neumann architecture, this new style of organising intermediate abstractions more "organically" could bring substantial flexibility to the toolchain to evolve together with the hardware landscape, and also improve the productivity of the library developers.

Appendices

Appendix A

Operators for experimental evaluation

Here we give the mathematical definitions of the bilinear forms $a(\cdot,\cdot)$ used as the test cases for the experimental evaluation.

mass Here u and v are scalar-valued trial and test functions.

$$a(u,v) = uv \; \mathrm{d}x \tag{A.1}$$

helmholtz Here u and v are scalar-valued trial and test functions.

$$a(u,v) = (\nabla u \cdot \nabla v \; + uv) \; \mathrm{d}x \tag{A.2}$$

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

$$a(\mathbf{u},\mathbf{v}) = (\nabla \mathbf{u} : \nabla \mathbf{v}) \; \mathrm{d}x \tag{A.3}$$

elasticity The linear elasticity model solves for a displacement vector field. Here $\mathbf{u}$ and $\mathbf{v}$ are vector-valued trial and test functions, ϵ is the symmetric strain rate tensor. The bilinear form is defined as:

$$\begin{aligned} \epsilon(\mathbf{u}) &= \frac{1}{2}\big[\nabla \mathbf{u} + (\nabla \mathbf{u})^T\big] \\ a(\mathbf{u},\mathbf{v}) &= \epsilon(\mathbf{u}) : \epsilon(\mathbf{v}) \; \mathrm{d}x \end{aligned} \tag{A.4}$$

mixed poisson Here we consider a mixed formulation of two coupled fields for the Poisson equation $\nabla^2 u = -f$. We introduce the the auxiliary vector-valued variable $\sigma = \nabla u$ to represent the negative flux. Let (σ, u) be the trial functions and (τ, v) be the test functions from a mixed function space of conforming discrete function spaces, the bilinear form is defined as:

$$a\left((\sigma,u),(\tau,v)\right) = \sigma \cdot \tau + \nabla \cdot \tau u + \nabla \cdot \sigma v \, \mathrm{d}x \tag{A.5}$$

hyperelasticity In this simple hyperelastic model, we define the strain energy function Ψ over vector field $\mathbf{u}$:

$$\begin{aligned}\mathbf{F} &= \mathbf{I} + \nabla \mathbf{u} \\ \mathbf{C} &= \mathbf{F}^T \mathbf{F} \\ \mathbf{E} &= (\mathbf{C} - \mathbf{I})/2, \\ \Psi &= \frac{\lambda}{2}\left[\mathsf{tr}(\mathbf{E})\right]^2 + \mu \mathsf{tr}(\mathbf{E}^2)\end{aligned} \tag{A.6}$$

where $\mathbf{I}$ is the identity matrix, λ and μ are the Lamé parameters of the material, $\mathbf{F}$ is the deformation gradient, $\mathbf{C}$ is the right Cauchy-Green tensor, $\mathbf{E}$ is the Euler-Lagrange strain tensor. We define the Piola-Kirchhoff stress tensors as:

$$\begin{aligned}\mathbf{S} &= \frac{\partial \Psi}{\partial \mathbf{E}} \\ \mathbf{P} &= \mathbf{F}\mathbf{S}\end{aligned} \tag{A.7}$$

Finally, we arrive at the residual form of this non-linear problem:

$$r = \mathbf{P} : \nabla \mathbf{v} - \mathbf{b} \cdot \mathbf{v} \tag{A.8}$$

where $\mathbf{b}$ is the external forcing. To solve this non-linear problem, we need to linearize the residual form at an approximate solution $\mathbf{u}$, this gives us the bilinear form a:

$$a(\delta \mathbf{u}, \mathbf{v}) = \lim_{\epsilon \to 0} \frac{r(\mathbf{u} + \epsilon \delta \mathbf{u}) - r(\mathbf{u})}{\epsilon}, \tag{A.9}$$

where the trial function is $\delta \mathbf{u}$, the test function is $\mathbf{v}$, and $\mathbf{u}$ is a coefficient of the operator. We use the automatic differentiation of UFL to compute the operator symbolically.

Bibliography

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, and Others. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells. Unified form language: A domain-specific language for weak formulations of partial differential equations. *ACM Transactions on Mathematical Software (TOMS)*, 40(2):9, 2014.

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

S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. D. Dalcin, V. Eijkhout, W. Gropp, D. Kaushik, and Others. Petsc users manual revision 3.8. Technical report, Argonne National Lab.(ANL), Argonne, IL (United States), 2017.

W. Bangerth, R. Hartmann, and G. Kanschat. deal. II—a general-purpose object-oriented finite element library. *ACM Transactions on Mathematical Software (TOMS)*, 33(4):24—-es, 2007.

D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. W. Scogland. RAJA: Portable performance for large-scale scientific applications. In *2019 ieee/acm international workshop on performance, portability and productivity in hpc (p3hpc)*, pages 71–81. IEEE, 2019.

J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM transactions on graphics (TOG)*, 22(3):917–924, 2003.

S. Brenner and R. Scott. *The mathematical theory of finite element methods*, volume 15. Springer Science and Business Media, 2007.

C. D. Cantwell, D. Moxey, A. Comerford, A. Bolis, G. Rocco, G. Mengaldo, D. De Grazia, S. Yakovlev, J.-E. Lombard, D. Ekelschot, and Others. Nektar++: An open-source spectral/hp element framework. *Computer physics communications*, 192:205–219, 2015.

C. Cecka, A. J. Lew, and E. Darve. Assembly of finite element methods on graphics processors. *International journal for numerical methods in engineering*, 85(5):640–669, 2011.

T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, and Others. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 578–594, 2018.

A. Dedner, R. Klöfkorn, M. Nolte, and M. Ohlberger. A generic interface for parallel and adaptive discretization schemes: abstraction principles and the DUNE-FEM module. *Computing*, 90(3):165–196, 2010.

A. Dziekonski, P. Sypek, A. Lamecki, and M. Mrozowski. Accuracy, memory, and speed strategies in GPU-based finite-element matrix-generation. *IEEE Antennas and Wireless Propagation Letters*, 11:1346–1349, 2012.

H. C. Edwards, C. R. Trott, and D. Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of parallel and distributed computing*, 74(12):3202–3216, 2014.

N. Fehn, W. A. Wall, and M. Kronbichler. A matrix-free high-order discontinuous Galerkin compressible Navier-Stokes solver: A performance comparison of compressible and incompressible formulations for turbulent incompressible flows. *International Journal for Numerical Methods in Fluids*, 89(3):71–102, 2019.

A. Fog. VCL — A C++ Vector Class Library. Technical report, 2017. URL https://www.agner.org/optimize/vectorclass.pdf.

M. B. Giles, G. R. Mudalige, B. Spencer, C. Bertolli, and I. Reguly. Designing OP2 for GPU architectures. *Journal of Parallel and Distributed Computing*, 73(11):1451–1460, 2013.

F. Hecht. New development in freefem++. *Journal of Numerical Mathematics*, 20(3-4):251–266, 2012.

J. Hoberock and N. Bell. Thrust: A parallel template library. 2010.

M. Homolya. On code generation techniques for finite elements. (June), 2018.

M. Homolya, R. C. Kirby, and D. A. Ham. Exposing and exploiting structure: optimal code generation for high-order finite element methods. *arXiv preprint arXiv:1711.02473*, 2017.

M. Homolya, L. Mitchell, F. Luporini, and D. A. Ham. TSFC: a structure-preserving form compiler. *SIAM Journal on Scientific Computing*, 40(3): C401—-C428, 2018.

C. T. Jacobs, S. P. Jammy, and N. D. Sandham. OpenSBLI: A framework for the automated derivation and parallel execution of finite difference solvers on a range of computer architectures. *Journal of Computational Science*, 18:12–23, 2017.

D. Kempf, R. Heß, S. Müthing, and P. Bastian. Automatic Code Generation for High-Performance Discontinuous Galerkin Methods on Modern Architectures. *arXiv preprint arXiv:1812.08075*, 2018.

Khronos Group. SYCL website. URL www.khronos.org/sycl/.

U. Kiran, S. S. Gautam, and D. Sharma. GPU-based matrix-free finite element solver exploiting symmetry of elemental matrices. *Computing*, 102(9):1941–1965, 2020.

R. C. Kirby. Algorithm 839: FIAT, a new paradigm for computing finite element basis functions. *ACM Transactions on Mathematical Software (TOMS)*, 30(4):502–516, 2004.

R. C. Kirby and L. Mitchell. Solver composition across the PDE/linear algebra barrier. *SIAM Journal on Scientific Computing*, 40(1):C76—-C98, 2018.

R. C. Kirby and K. T. Thinh. Fast simplicial quadrature-based finite element operators using Bernstein polynomials. *Numerische Mathematik*, 121(2): 261–279, 2012.

I. Kiss, S. Gyimothy, Z. Badics, and J. Pavo. Parallel realization of the element-by-element FEM technique by CUDA. *IEEE Transactions on magnetics*, 48(2):507–510, 2012.

A. Klöckner. 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*, page 82. ACM, 2014.

A. Klöckner, T. Warburton, J. Bridge, and J. S. Hesthaven. Nodal discontinuous Galerkin methods on graphics processors. *Journal of Computational Physics*, 228(21):7863–7882, 2009.

A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Computing*, 38(3):157–174, 2012. ISSN 0167-8191. doi: 10.1016/j.parco.2011.09.001.

M. G. Knepley and D. A. Karpeev. Mesh algorithms for PDE with sieve I: Mesh distribution. *Scientific Programming*, 17(3):215–230, 2009. ISSN 10589244. doi: 10.3233/SPR-2009-0249.

M. G. Knepley and A. R. Terrel. Finite element integration on GPUs. *ACM Transactions on Mathematical Software (TOMS)*, 39(2):10, 2013.

M. G. Knepley, K. Rupp, and A. R. Terrel. Finite element integration with quadrature on the GPU. *arXiv preprint arXiv:1607.04245*, 2016.

D. Komatitsch, D. Michéa, and G. Erlebacher. Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA. *Journal of Parallel and Distributed Computing*, 69(5):451–460, 2009.

M. Kronbichler and K. Kormann. Fast matrix-free evaluation of discontinuous Galerkin finite element operators. *arXiv preprint arXiv:1711.03590*, 2017.

M. Lange, L. Mitchell, M. G. Knepley, and G. J. Gorman. Efficient mesh management in firedrake using PETSC DMPLEX. *SIAM Journal on Scientific Computing*, 38(5):S143—-S155, 2016.

C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. MLIR: A compiler infrastructure for the end of Moore's law. *arXiv preprint arXiv:2002.11054*, 2020.

C. Lengauer, S. Apel, M. Bolten, A. Größlinger, F. Hannig, H. Köstler, U. Rüde, J. Teich, A. Grebhahn, S. Kronawitter, and Others. ExaStencils: Advanced stencil-code engineering. In *European Conference on Parallel Processing*, pages 553–564. Springer, 2014.

K. Ljungkvist. Matrix-free finite-element computations on graphics processors with adaptively refined unstructured meshes. In *SpringSim (HPC)*, page 1, 2017.

A. Logg, K.-A. Mardal, and G. Wells. *Automated solution of differential equations by the finite element method: The FEniCS book*, volume 84. Springer Science and Business Media, 2012.

F. Luporini, A. L. Varbanescu, F. Rathgeber, G.-T. Bercea, J. Ramanujam, D. A. Ham, and P. H. J. Kelly. Cross-loop optimization of arithmetic intensity for finite element local assembly. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):57, 2015.

F. Luporini, D. A. Ham, and P. H. J. Kelly. An algorithm for the optimization of finite element integration loops. *ACM Transactions on Mathematical Software (TOMS)*, 44(1):3, 2017.

F. Luporini, M. Lange, M. Louboutin, N. Kukreja, J. Hückelheim, C. Yount, P. Witte, P. Kelly, F. Herrmann, and G. Gorman. Architecture and performance of Devito, a system for automated stencil computation. *CoRR*, abs/1807.0, jul 2018. URL `http://arxiv.org/abs/1807.03032`.

G. R. Markall, D. A. Ham, and P. H. J. Kelly. Towards generating optimised finite element solvers for GPUs from high-level specifications. *Procedia Computer Science*, 1(1):1815–1823, 2010.

J. Martinez-Frutos, P. J. Martinez-Castejon, and D. Herrero-Perez. Fine-grained GPU implementation of assembly-free iterative solver for finite element problems. *Computers & Structures*, 157:9–18, 2015.

J. D. McCalpin. HPL and DGEMM performance variability on the Xeon Platinum 8160 processor. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 18. IEEE Press, 2018.

A. T. T. McRae, G.-T. Bercea, L. Mitchell, D. A. Ham, and C. J. Cotter. Automated generation and symbolic manipulation of tensor product finite elements. *To appear in SIAM Journal on Scientific Computing*, pages 1–29, 2014. ISSN 10957200. doi: 10.1137/15M1021167.

D. Moxey, R. Amici, and M. Kirby. Efficient matrix-free high-order finite element evaluation for simplicial elements. *SIAM Journal on Scientific Computing*, 42(3):C97—-C123, 2020.

G. R. Mudalige, M. B. Giles, I. Reguly, C. Bertolli, and P. H. J. Kelly. Op2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. In *2012 Innovative Parallel Computing (InPar)*, pages 1–12. IEEE, 2012.

G. R. Mudalige, I. Z. Reguly, and M. B. Giles. Auto-vectorizing a large-scale production unstructured-mesh CFD application. In *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*, page 5. ACM, 2016.

E. Müller, X. Guo, R. Scheichl, and S. Shi. Matrix-free GPU implementation of a preconditioned conjugate gradient solver for anisotropic elliptic PDEs. *Computing and Visualization in Science*, 16(2):41–58, 2013.

E. H. Müller, R. Scheichl, and E. Vainikko. Petascale solvers for anisotropic PDEs in atmospheric modelling on GPU clusters. *Parallel Computing*, 50:53–69, 2015.

S. Müthing, M. Piatkowski, and P. Bastian. High-performance Implementation of Matrix-free High-order Discontinuous Galerkin Methods. *arXiv preprint arXiv:1711.10885*, 2017.

K. B. Ølgaard and G. N. Wells. Optimisations for quadrature representations of finite element tensors through automated code generation. *ACM Transactions on Mathematical Software (TOMS)*, 37(1):8:1–8:23, 2010. ISSN 0098-3500. doi: 10.1145/1644001.1644009.

OpenACC Organization. OpenACC website. URL www.openacc.org.

OpenMP Architecture Review Board. OpenMP Application Programming Interface 5.0. Technical Report November, OpenMP Architecture Review Board, 2018. URL https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

F. Pichler and G. Haase. Finite element method completely implemented for graphic processor units using parallel algorithm libraries. *The international journal of high performance computing applications*, page 1094342017694703, 2017.

L. Ram and D. Sharma. Evolutionary and GPU computing for topology optimization of structures. *Swarm and evolutionary computation*, 35:1–13, 2017.

F. Rathgeber, G. R. Markall, L. Mitchell, N. Loriant, D. A. Ham, C. Bertolli, and P. H. J. Kelly. PyOP2: A high-level framework for performance-portable simulations on unstructured meshes. *Proceedings - 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, SCC 2012*, pages 1116–1123, 2012. doi: 10.1109/SC.Companion.2012.134.

F. Rathgeber, D. a. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. McRae, G.-T. Bercea, G. R. Markall, and P. H. J. Kelly. Firedrake: automating the finite element method by composing abstractions. 0(0): 1–25, 2015. URL http://arxiv.org/abs/1501.01809.

K. Rupp, P. Tillet, F. Rudolf, J. Weinbub, A. Morhammer, T. Grasser, A. Jungel, and S. Selberherr. ViennaCL—linear algebra library for multi- and many-core architectures. *SIAM Journal on Scientific Computing*, 38 (5):S412—-S439, 2016.

T. Sun. tj-sun/firedrake-vectorization: Scripts for experimental evaluation for the manuscript on cross-element vectorization., mar 2019a. URL https://doi.org/10.5281/zenodo.2590705.

T. Sun. Cross-element vectorization in Firedrake. https://www.codeocean.com/, feb 2019b.

T. Sun, L. Mitchell, K. Kulkarni, A. Klöckner, D. A. Ham, and P. H. J. Kelly. A study of vectorization for matrix-free finite element methods. *The International Journal of High Performance Computing Applications*, 34(6):629–644, 2020.

The OpenFOAM Foundation. OpenFOAM user guide Version 6. *The OpenFOAM Foundation*, 2018.

N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

S. Verdoolaege. isl: An integer set library for the polyhedral model. In *International Congress on Mathematical Software*, pages 299–302. Springer, 2010.

S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, pages 65–76, 2009. URL http://dl.acm.org/citation.cfm?id=1498785.

F. D. Witherden, A. M. Farrington, and P. E. Vincent. PyFR: An open source framework for solving advection–diffusion type problems on streaming architectures using the flux reconstruction approach. *Computer Physics Communications*, 185(11):3028–3040, 2014.

Zenodo/Firedrake. Firedrake: an automated finite element system, mar 2019. URL https://doi.org/10.5281/zenodo.2590703.

B. Zhang. Guide to automatic vectorization with Intel AVX-512 instructions in Knights Landing processors. *Colfax International¡ http://colfaxresearch. com*, 2016.