Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Optimised small-matrix linear algebra in Firedrake

Author:
Natalia Arciuchiewicz

Supervisor:
Paul Kelly

Second Marker:
David Ham

June 17, 2020

Abstract

Many physical processes can be modelled and understood using partial differential equations, such as fluid flow, cell growth and structural analysis.

Firedrake is a compiler for a domain-specific language. Given a partial differential equation and a discretisation method, it uses finite elements to generate code to approximately solve the equation.

The aim of this project is to optimise operations on local, small-matrix tensors in Firedrake so that they are more efficient in terms of the memory and number of calculations required. This was achieved in two ways:

Firstly, by **reducing the amount of memory needed by temporaries** to perform operations on tensors. This was done through smart indexing, which was implemented through manipulations of the expression graph. Although the optimisation still created some unneccessary memory associated with temporaries, it did reduce the amount of memory needed by 55% - 65%.

A model was developed to predict the amount of memory needed by temporaries before and after our changes, based on things like mesh and operation order. The model was applied to various examples, including the common and useful example of hybridization, with predictions validated by comparison with the true values obtained from the output code.

Secondly, by **taking advantage of the sparsity of local tensors**. Sparse local tensors were successfully implemented via a buffer of non-zero values and a mapping of these back into the full tensor.

We developed a solution to show how PyOP2 global assembly could be adapted to support sparse tensors and demonstrated its correctness through a proof of concept. We deduced that by completing the implementations of sparse local tensors, we could reduce the memory and calculation on the local tensors to by over 50% for 2D tensors and over 66% for 3D tensors.

Acknowledgements

I would like to thank everyone who has supported me throughout this project.

In particular, I thank my supervisors Paul Kelly and David Ham for their support and advice. I greatly appreciated the frequent meetings and interest in the success of the project. Their knowledge and dedication have been invaluable.

I also thank Sophia Vorderwuelbecke for her support and patience in helping me understand the system and overcome issues.

I am grateful to all those who read this thesis and especially Andrew Young for his inspirational discussions, feedback and support which helped and encouraged me throughout.

Finally, thank you to my family and friends for their encouragement throughout my time at Imperial.

Contents

Chapter 1

Introduction

This thesis presents optimisations that allow local, small-matrix tensors in Firedrake to be used more efficiently in terms of memory and calculation required. The contributions fall into two main categories: (1) reducing the number of temporaries needed to perform manipulations on the tensors, and (2) taking advantage of the sparsity of local tensors.

In this chapter, we outline the context of this project, and hence show the importance of the work for solving real-life problems. We discuss the objectives of the project and highlight our key contributions.

1.1 Context: Automating the finite element method

Many physical processes, such as fluid flow or cell growth, can be modelled and understood using partial differential equations (PDEs). The finite element method (FEM) is a widely used numerical method for solving PDEs. There are many tools that automate the process of using FEM to solve problems. Notable among these is Firedrake, on which this project is based.

Why is the finite element method widely used?

FEM can be a powerful technique of finding solutions to interesting and useful questions. It uses discretisation to divide the system into finite elements. This is beneficial since it allows for an accurate representation of complex geometry, capturing of local effects and a simple representation of the whole solution. [12] Advantages of FEM include: [13, 14, 15]

- Modelling objects made from many different materials.
- The size of the elements can be adapted for better representations.
- Can be used for time-dependent simulations.
- Visualisations can easily be created from it.

FEM has many strengths, but in order to take advantage of these, we need to find a way to make it easy to use. This leads us on to our next question.

How can we use the finite element method?

It is possible to hand-calculate the results of problems using FEM, but this is very cumbersome. Many steps in FEM can be automated by a computer, and luckily the mathematics of FEM can often be simply and abstractly specified.

However, it is difficult and time consuming to hand-code efficient, parallel and correct implementations. A key disadvantage of hand-coding low-level code is the loss of abstraction. When working on such low level code, where all the implementations of the different parts of the process are not separated, it is much harder to reason about the program and the higher-level mathematical concepts, as well as to make changes to the choice of a certain part, e.g. wanting to solve a different PDE.

All this naturally leads us to create tools that automate the finite element method that take in a user-defined problem and automatically generate code to solve the problem using FEM. Many such software solutions have been implemented, including the system used in this project: Firedrake [5].

Why use Firedrake as opposed to other systems utilising the finite element method?

What differentiates Firedrake from other finite element system is that it has fine-grain separation of the FEM abstraction. By separating the calculation into many specialised layers, Firedrake allows people knowledgable about different disciplines to add value within their domain of expertise without needing to have much understanding of the other parts of the systems or the underlying mathematics. In addition, layers allow optimisations to automatically and easily be performed on many different levels of abstraction.

Firedrake users use Unified Form Language (UFL) to specify the mathematics of their problems. UFL is a domain specific language (DSL) embedded in Python that allows the expression of FEM problems compactly and in a language close to mathemetics. This makes it more intuitive and simple to use, according to its authors Alnaes et al [16]. With UFL, users can easily specify their problems without needing to think about the lower-level implementations.

However, Firedrake is not fully optimised for all FEM problems. Two methods have become increasingly popular, especially in simulating geophysical flows. These are discontinuous Galerkin methods and mixed FEMs. It is difficult to create solvers for the systems these methods produce; it can be more efficient to solve the systems by directly manipulating local tensors.

How can we manipulate local tensors in Firedrake?

Slate is an embedded DSL in Firedrake that can be used to manipulate local tensors. [4]

How do we efficiently manipulate local tensors in Firedrake?

This project aims to address this question.

1.2 Objectives

The objective of this project is to optimise the layer in Firedrake responsible for tensor computations. This is accomplished through the following:

1. Reducing the amount of memory needed by temporaries to perform operations on tensors.
2. Taking advantage of the sparsity of local tensors to reduce memory and computation.

1.3 Contributions

This thesis has the following contributions:

1. A new optimisation phase is proposed that reduces the number of temporaries through smart indexing, and the implementation of this in Firedrake through manipulations of the expression graph. This has led to a reduction of the memory required for temporaries by 55%-65%.
2. Analysis of the stacking effects of different operations, i.e. how the number of temporaries needed for a sequence of operations is different to what one may expect from taking the operations in isolation.
3. A model for the memory required for temporaries before and after our changes, based on things like the mesh and the finite element; application of the model to various examples, including the common and useful example of hybridization, validated through comparison with the true values obtained from the output code.
4. Design and implementation of a new optimisation in Firedrake that takes advantage of the sparsity of local tensors. For 2D tensors, this cuts the space and calculation to just over 50%, while for 3D tensors these are cut by almost 66%.
5. Analysis of global assembly in PyOP2 and proposal for how it could be changed to support structurally sparse local tensors.

1.4 Thesis structure

In chapter 2 we provide a review of relevant literature that forms the background knowledge related to this project and related work.

In chapter 3 we discuss the work undertaken in this project with relation to reducing the amount of memory needed for temporaries, giving a run through of the ideas and approaches along the way.

In chapter 4 we evaluate our work on reducing the memory for temporaries, putting it into context of similar work and analysing the effects of our changes through the creation of a model to predict memory required for temporaries. We then use this model on several examples, most notably on a hybridization example.

In chapter 5 we present our work on implementing structured sparse local tensors, including a discussion about the limitations of PyOP2 with regards to performing global assembly on the local sparse tensors.

Finally, in chapter 6 we conclude the project, discussing its successes, limitations in the current Firedrake system, and possible future work.

Chapter 2

Background

This chapter is intended to familiarise the reader with the Firedrake toolchain and to give some context for the work presented later in this thesis. This requires a brief outline of the mathematical concepts related to the finite element method. We then go on to discuss related work in order to show Firedrake in the context of similar systems.

A summary of commonly used acronyms is included in the appendix. The appendix also includes a glossary with items not defined explicitly in the text that we have assumed the reader is familiar with.

2.1 Finite element method

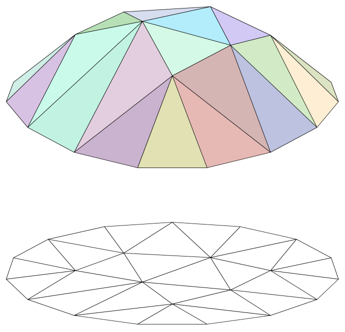


Figure 2.1: An example of a mesh [12]

Partial differential equations (PDEs) are either elliptic, hyperbolic or parabolic. The common way of solving elliptic PDEs is using variational methods, which includes the finite element method (FEM). [17]

The finite element method is a type of numerical method for solving PDEs. Given a differential equation, the finite element method generates an algorithm for approximating the solution. [2] FEM uses discretisation implemented by a 'mesh' (e.g. 2.1) to divide the system into 'finite elements'. Computations are performed on each element and the results of these are assembled into a system of polynomial equations which are then solved. [12]

We will give a general introduction to the steps in FEM through a simple example, given below.

2.1.1 The steps in FEM: A simple example [1, 2, 3]

Before diving in to, we briefly discuss an important concept that will be used in this example.

Boundary conditions Boundary conditions are enforced to ensure a unique solution to the differential equation. There are two main forms of boundary conditions, each enforced in a different way in the finite element method: [1, 2]

- Dirichlet (e.g. u(0) = 0), which specify the value of the solution on the boundary. Also called 'essential' since they are in the essence of the problem and appear in the weak form explicitly (discussed later).
- Neumann (e.g. u'(1) = 0), which specify the values of the derivatives within the boundary of the domain. Also called 'natural' since they naturally emerge from the problem. They are incorporated implicitly in the weak form.

Let's now consider a simple example to demonstrate the steps in FEM, namely Poisson's equation in one dimension:

$$-\nabla^2 u = f, \text{ in } [0,1] \tag{2.1}$$

$$\text{where } u(0) = 0, \tag{2.2}$$

$$u'(1) = 0 \tag{2.3}$$

Here u is an unknown function we wish to find that we call the *trial function*. Let's walk through the steps of FEM to find u.

Step 0: Strong form. We start with the PDE in the *strong form*. For our example, this is equation 2.1. In this form, the equality in 2.1 applies for every point in the domain and the boundary conditions apply for the boundary points. Often PDEs in the strong form cannot be solved as-is. If we are willing to accept a wider class of solutions, such as functions that are not everywhere continous, we can transform the PDE into a "weaker" form in order to solve it.

Step 1: Weak/Integral form. The *weak form*, also called the *integral form*, is a reformulation of the original (strong) form of the equation which has much weaker constraints on valid solutions. The solutions to the weak form are a superset of the solutions to the strong form.

To bring the equation into this weaker form, we weigh the equation with a (sufficiently regular) *test function* v, integrate over the entire domain, then integrate by parts.

We perform integration by parts in order to lower the order of derivatives (we want $u'(x)$, not $u''(x)$), as this lowers the order of continuity required. Let us define the space V of suitable test functions, v, which includes the conditions that must be imposed on v.

$$V = \{v \in L^2([0,1]) : a(v,v) < \infty \text{ and } v(0) = 0\} \tag{2.4}$$

where the constraint that $v(0) = 0$ comes from the Dirichlet condition in the original problem.

Putting this all together we get:

$$(f,v) := \int_0^1 fv \,\mathrm{d}x = \int_0^1 -u''v \,\mathrm{d}x = \int_0^1 u'v' \,\mathrm{d}x - u'v|_0^1 = \int_0^1 u'v' \,\mathrm{d}x =: a(u,v) \tag{2.5}$$

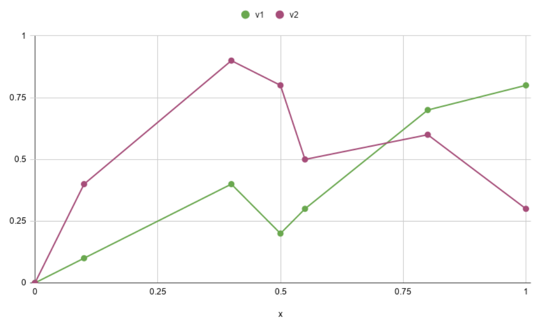
In the above workings, $u'v|_0^1 = 0$ because of the boundary conditions of our original problem 2.1. This is what we mean by the Neumann boundary conditions being incorporated implicitly in the weak form.

Then the *variational* (*weak*) form of 2.1 is:

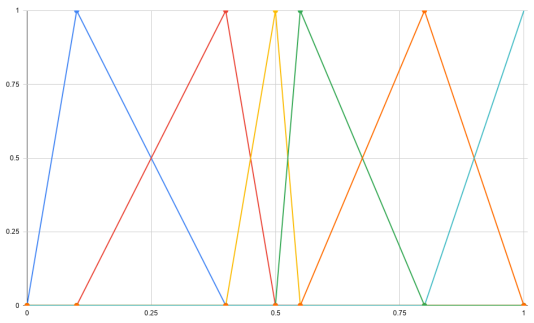
$$u \in V \text{ such that } a(u,v) = (f,v)\ \forall v \in V \tag{2.6}$$

This form is also called the *variational form* because v can vary over all possible values. If $f \in C^0([0,1])$ and $u \in C^2([0,1])$ (i.e. the functions are sufficiently well-behaved), and f and u satisfy the weak form 2.6, then u solves the original equation 2.1 [1]. Here we use $C^n([0,1])$ to denote the space of functions with n continuous derivatives on the interval $[0,1]$.

In the weak form, we are still looking for an infinite-dimensional solution in an infinite-dimensional space. We want to make our problem finite-dimensional so that we can implement it in code.



(a) Example of two functions, v1 and v2, that are in our chosen S.



(b) Basis for our chosen S.

Step 2: Discretization of the weak form. We choose a finite-dimensional subspace, call it S, of the function space we are in, $S \subset V$. Hence, we now seek

$$u_s \in S \text{ such that } a(u_s, v) = (f, v)\ \forall v \in S \tag{2.7}$$

Here we see that we have excluded some possible solutions for u, but we have also reduced the number of test functions v we need to look at. If $f \in L^2([0,1])$ then 2.7 has a unique solution (proof omitted).

Let's look at a possible space S and a mesh of it for our one-dimensional Poisson example. Consider $0 = x_0 < x_1 < ... < x_n = 1$ an ordered set of points in our domain $[0,1]$ that we will call the *nodes* of our system. The intervals between these points, $[x_{i-1}, x_i]$, form a partition of the entire domain $[0,1]$. These intervals are called the *elements* and together they form a *mesh* on the domain. We can define S to be the linear space of functions v such that:

1. $v \in C^0([0,1])$
2. $v|_{[x_{i-1},x_i]}$ is a linear polynomial for $i = 1, ..., n$ (i.e. of the form $y = ax + b$)
3. $v(0) = 0$ (the boundary condition from the original definition of the PDE in equation 2.1)

Let's arbitarily for this example choose the nodes to be point evaluation at

$$x = \{0, 0.1, 0.4, 0.5, 0.55, 0.8, 1\} \tag{2.8}$$

An example of two functions, v1 and v2, that would be part of S for nodes 2.8 is shown in figure 2.2a.

The errors in FEM come from the difference between the discretised space S and the continuous space V. The error decreases when the mesh is refined i.e. when we add in more nodes to make the diameter of each element smaller.

Step 3: Assembly procedure. Now we use the assembly procedure to turn 2.7 into a system of linear equations that we can then solve. In short, the assembly procedure consists of two main steps:

1. *Local assembly.* Compute local $n \times n$ matrix K^e and local $n \times 1$ vector F^e for each element e in the domain, where n is the number of nodes per element.
2. *Global assembly.* Form global matrix K and global vector F from the local ones.

Let's work through these steps in more detail with our Poisson example. We first introduce some concepts needed for the assembly steps.

We must decide on a basis for our discrete space S. A possible basis for S for our Poisson example is the set

$$\{\phi_i : \phi_i(x_j) = \delta_{ij}\ i = 1, ..., n\} \tag{2.9}$$

We illustrate this *global basis* in figure 2.2b, where we've taken the same set of nodes as before, 2.8.

Since we are computing locally then combining globally, we need a way to translate between local and global node references. This is done via a *global-to-local* index (also called a *cell-node map* or a *facid node map*). This index, which we will denote $i(e, j)$, gives the global node number of local node j, given that we are in element e. For our Poisson example, each element corresponds to two nodes, so $j \in \{0, 1\}$. For example, if we are in element $[0.1, 0.4]$ then $i(e, 0) = 1$ and $i(e, 1) = 2$.

The change of basis from global $x \in [0, 1]$, which moves across all elements, to local $X \in [0, 1]$, which moves across a single element $[x_{j-1}, x_j]$, will be

$$x = x_{j-1} + X(x_j - x_{j-1}) \tag{2.10}$$

Let us define an *interpolant* $v_I \in S$ of $v \in C^0([0, 1])$ to be determined by $v_I := \sum_{i=1}^{n} v(x_i)\phi_i$. We can write interpolant functions for functions in space S as:

$$f_I = \sum_e \sum_{j=0}^{1} f(x_{i(e,j)})\phi_j^e \tag{2.11}$$

where $\{\phi_j^e : j = 0, 1\}$ denotes the local basis function for a given element $[x_{e-1}, x_e]$:

$$\phi_j^e(x) = \phi_j \left(\frac{x - x_{e-1}}{x_e - x_{e-1}} \right) \tag{2.12}$$

where

$$\phi_0(x) := \begin{cases} 1 - x & \text{if } x \in [0, 1] \\ 0 & \text{otherwise} \end{cases} \quad \text{and} \quad \phi_1(x) := \begin{cases} x & \text{if } x \in [0, 1] \\ 0 & \text{otherwise} \end{cases} \tag{2.13}$$

At this point we've decided on a global basis, an index to translate between local and global node references, defined the change of basis mapping, determined a local basis, and have a way of interpolating continuous functions in our discrete space S. Now we can assemble (i.e. compute) equation 2.7:

$$u_s \in S \text{ such that } a(u_s, v) = (f, v) \; \forall v \in S \tag{2.14}$$

Let's look at the right hand side first: (f, v). Here v is our test function. Fortunately, testing just the basis functions is sufficient because then we've tested with all the functions in the function space (since any function $v \in S$ can be represented as $v = \sum_i v_i\phi_i$ and by linearity of $a(u_s, v)$ and (f, v)).

Since we are working in a discrete space we want to represent f discretely so we re-write it as a linear combination of some basis elements (usually, but not necessarily, the same as the basis for S). For simplicity, in this example let's use the same basis elements as for S. Hence our continuous f can be represented by the interpolant form given in 2.11.

Putting it all together, for the right hand side we get:

$\forall i = 1, ..., n$ and $\forall v \in S$:

$$(f, v) \equiv (f, \phi_i) := \int_0^1 f\phi_i \, \mathrm{d}x \qquad \text{(Sufficient to test only basis functions)} \tag{2.15}$$

$$\equiv \sum_e \int_e f\phi_i \, \mathrm{d}x \qquad \text{(Summing all local element contributions)} \tag{2.16}$$

$$= \sum_e \sum_{j=0}^{1} f(x_{i(e,j)}) \int_e \phi_i\phi_j^e \, \mathrm{d}x \qquad \text{(Represent f discretely)} \tag{2.17}$$

Changing 2.17 into local coordinates we see that for each element we want to find F^e such that:

$$F^e = \sum_{j=0}^{1} f(x_{i(e,j)}) \int_e \phi_i^e\phi_j^e \frac{\mathrm{d}x}{\mathrm{d}X} \mathrm{d}X \qquad \forall i = 1, ..., n \tag{2.18}$$

where

$$\frac{\mathrm{d}x}{\mathrm{d}X} = x_{i(e,1)} - x_{i(e,0)} \tag{2.19}$$

from our change of basis 2.10. Note that $x_{i(e,1)} - x_{i(e,0)}$ is just the length of the interval.

Since $j \in \{0, 1\}$, 2.18 gives us back two values corresponding to the local contributions of the current element to the global nodes associated with it (e.g. if the current element is $[0.1, 0.4]$ then this calculation will give us the local contributions of this element to global nodes 1 and 2).

Let's now look at the left hand side. We have that $\forall i = 1, ..., n$ and $\forall v \in S$:

$$a(u_s, v) = a(\sum_{j=1}^{n} u_j \phi_j, v) = \sum_{j=1}^{n} u_j a(\phi_j, v) \qquad \text{(Substitute } u_s = \sum_{j=1}^{n} u_j \phi_j\text{)} \tag{2.20}$$

$$\equiv \sum_{j=1}^{n} u_j a(\phi_j, \phi_i) \qquad \text{(Sufficient to test only basis functions)} \tag{2.21}$$

$$=: \sum_{j=1}^{n} u_j K^e \tag{2.22}$$

Taking K^e from 2.22 and changing it into local coordinates as well as changing the basis for the derivatives appropriately we see that from each element we will get $\forall i \in \{0, 1\}$ and $\forall j \in \{0, 1\}$

$$K^e = \int_0^1 \frac{\mathrm{d}X}{\mathrm{d}x} \frac{\mathrm{d}\phi_j^e(X)}{\mathrm{d}X} \frac{\mathrm{d}X}{\mathrm{d}x} \frac{\mathrm{d}\phi_i^e(X)}{\mathrm{d}X} \frac{\mathrm{d}x}{\mathrm{d}X} \mathrm{d}X \tag{2.23}$$

Here i and j can both take two values hence K^e will be a 2×2 matrix.

Typically, the local assembly matrix K^e is evaluated approximately using *numerical quadrature*. Using numerical quadrature to find, for a domain $e \subset \mathbb{R}^n$, the integral of a suitable function f (i.e. f is never discontinuous inside e) involves the summation of f evaluated on specifically chosen *quadrature points* and then weighted appropriately:

$$\int_e f(x) \mathrm{d}x = \sum_i w_i f(x_i) + \mathrm{O}(|e|^k) \tag{2.24}$$

The second term is the error term, which can be reduced to a desired level by using more quadrature points.

The weights depend on the diameter of each element, call this h. Practically, we want to determine the combination of the quadrature points and the weights, collectively known as *quadrature rules*, as arrays of numbers independent of h. To do this, we determine these quadrature rules for a *reference element*. Then when we want to perform numerical quadrature on our mesh elements, we can change coordinates to our reference element to perform the computation there. The change of basis is found using the Jacobian. [18]

The global matrix K is composed out of all the individual K^e elements. For example, when calculating K^e for element $[0.1, 0.4]$, K's 1st and 2nd row and column would be affected. The sparsity of K depends on the connectivity of the mesh, hence K is typically very sparse.

Step 4: Solve the resulting system of linear equations. Putting this all together, we have $u_s = \sum_{j=1}^{n} u_j \phi_j$. For each element, we also have $K^e = a(\phi_j, \phi_i)$ and $F^e = (f, \phi_i)$.
Using these parts, let's build the system of linear equations:

- Let $\boldsymbol{u} = (u_j)$
- Assemble all the local contributions of the K^e elements into $\boldsymbol{K}$
- Assemble all the local F^es into $\boldsymbol{F}$.

We now see that equation 2.7 is equivalent to

$$\boldsymbol{K}\boldsymbol{u} = \boldsymbol{F} \tag{2.25}$$

In this system of linear equations, the basis functions for S are the *test functions* we are testing, where each test function corresponds to one row in the *stiffness* matrix, $\boldsymbol{K}$.
We can solve this system of linear equations using variational or iterative methods.
We note the properties of K as: symmetric, positive definite and typically sparse.
[1, 2, 3]

2.1.2 More details about FEM

Things that can change between finite element methods These are: [12]

- a variational formulation (e.g. Galerkin, discontinuous Galerkin, mixed methods)
- a discretisation strategy. This involves creating the mesh, choosing the shape functions (see 2.1.3) and defining the mapping between the reference element and the physical elements of the mesh.
- solution algorithms. These are classified as either direct or iterative.
- post-processing steps. These can be used to extract information of interest from the solution as well as to perform error estimation.

Galerkin methods (Continuous) Galerkin methods are methods that take continuous operator problems, change them into a weak formulation, discretize, then apply the specified constraints on the discrete function space. Most relevantly here, an example of such a method is the Galerkin method of weighted residuals which is often used to find the global stiffness matrix in FEM. The solutions of this example method are represented as weighted sums of test functions, where the method seeks to find the weights such as to minimize the error between the actual solution and the approximation given by the test function combination. [19, 20]

Discontinuous Galerkin (DG) methods are similar to continuous Galerkin methods, but DG supports trial spaces that are only piecewise continous; thus DG has no continuity constraints across element boundaries. [21] To work around the lack of continuity, the connection between cells is defined by flux terms that are represented as integrals over the *interior facets* (*jump conditions*). [11]

2.1.3 FEM implementation

Here we discuss some key ideas for understanding how the finite element method can be implemented in code.

Finite element The concept of a *finite element* was formalised by Ciarlet in 1978 as: [1]

1. $K \subset \mathbb{R}^n$ the element domain, the *cell*. This should be a bounded and closed set, have a non-empty interior and have a piecewise smooth boundary. This is just the shape to define the element on.
2. P is the space of functions $K \to \mathbb{R}^n$. Refered to as the shape functions, this is the finite dimensional space of functions on K. This is just the polynomials the element should be able to represent.
3. $N = \{N_1, N_2, ..., N_k\}$ a basis for P^*, the dual of P. These are the nodal variables, also called the set of *degrees of freedom*. This is how one specifies "where" the solutions should live.

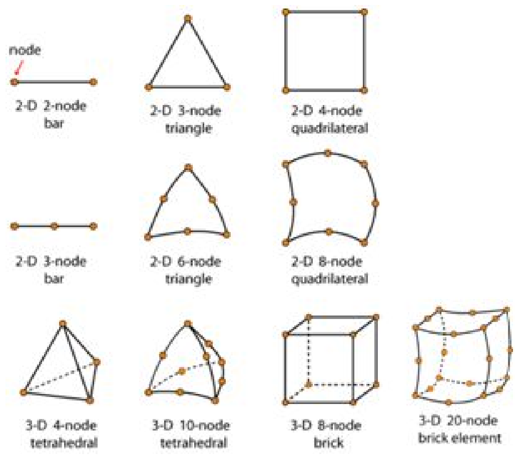
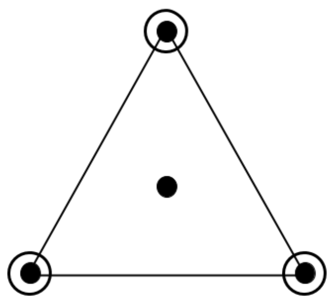
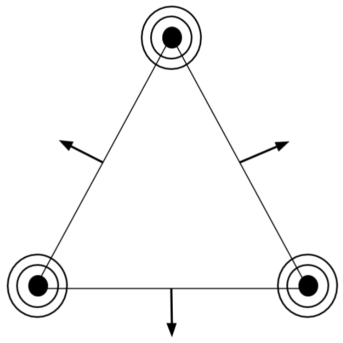


Figure 2.3: Examples of finite elements [22]

Nodal basis Letting $N = \{\phi_j^*\}_j$ be a basis for P^*, then a *nodal basis*, $\{\phi_i\}_i$ for P is a basis for P with the property that $\phi_j^*(\phi_i) = \delta_{ij}$. Examples of nodal bases include: [18]

- evaluation of the function $f \in P$ at points on the cell.
- evaluating the integral of the function over e.g. a cell, vertex, edge, face.
- evaluating the gradient of the function at a given point.


(a) Cubic hermite element on a triangle.


(b) Quintic Argyris element on a triangle.

Nodes and degrees of freedom We briefly note the difference between nodes and degrees of freedom. The degrees of freedom are the number of all the things you can evaluate. A node is a point in the domain to which degrees of freedom are assigned. A particular node can have multiple degrees of freedom. For example, a node on a vertex could calculate the function at the point and also the function's gradients in two directions – one node, three degrees of freedom. [23, 24]

Nodal basis depictions We briefly introduce the conventional depictions of common nodal bases on a 2D triangle. We use the cubic Hermite (2.4a) and quintic Argyris element (2.4b) as a reference. The '•' denotes the nodal variable evaluation at that point. '◯' denotes evaluation of the gradient at the given point (note that the gradient can have multiple components so this may give more than one degree of freedom). '◎' denotes evaluation of the Hessian. The outgoing arrows denote evaluation of the gradient normal to the three triangle edges. [18]

2.2 Hybridization, Static Condensation, Local Post-Processing

Here we briefly discuss some methods related to FEM which the reader should be familiar with in order to appreciate the capabilities of Slate, as well as to understand the choice of FEM problems used in the later parts of this paper.

2.2.1 Background mathematical concepts

We begin by providing definitions and brief descriptions of mathematical concepts that are useful for understanding hybridization, static condensation and local post-processing.

Lagrange multipliers In summary, the method of Lagrange multipliers seeks to find local extrema of function $f(x)$ subject to equality constraint $g(x) = 0$. It does this by finding the stationary points of the Lagrangian function $L(x, \lambda) = f(x) - \lambda g(x)$, with λ referred to as the Lagrange multiplier. [25]

2.2.2 Hybridization [4]

Hybridization is a method that takes discrete equations, and transforms them into a form to which one can apply the static condensation and local post-processing methods, such that the systems can be solved more easily. Hybridization methods can be used to avoid the difficult task of making sparse approximations of dense elliptical operators.

The challenge with implementing hybridization is making the code efficient and able to handle complex models, as well as keeping separate the different levels of abstraction. Slate is Firedrake's solution to this challenge.

In summary, what hybridization does is create an equivalent but discontinuous variant of the original problem, leading to the coupling of velocity only within cells. Within this process, Lagrange multipliers are introduced in order to enforce certain continuity constraints.

We will use the following example of an elliptical equation as an example of hybridization of mixed methods. This example is an adaptation of the example found in Gibson 2018 [4].

Notation

- Take dx and ds to be the standard measures of integration.
- $\Omega \subset \mathbb{R}^n$, the domain in which we are working.
- T_h the tesselation of Ω, made up of polygonal elements K associated with mesh size parameter h.
- $\partial T_h = \{e \in \partial K : K \in T_h\}$ the set of factes of T_h
- The jump of the normal component of any double-valued vector field w across a facet $e \in \partial T_h$ is:
$$[[w]]_e = \begin{cases} w|_{e^+} \cdot \mathbf{n}_{e^+} + w|_{e^-} \cdot \mathbf{n}_{e^-}, & e \in \partial T_h \setminus \partial\Omega \\ w|_e \cdot \mathbf{n}_e, \quad e \in \partial T_h \cap \partial\Omega \end{cases} \tag{2.26}$$
with $+$ and - denoting arbitary but globally defined sides of the facet.

Bringing a mixed method example into a solvable form

$$-\nabla \cdot (\kappa \nabla p) + cp = f, \text{ inside the domain } \Omega \tag{2.27}$$
$$p = p_0, \text{ on } \partial\Omega_p \tag{2.28}$$
$$-\kappa \nabla p \cdot \mathbf{n} = g, \text{ on } \partial\Omega N \tag{2.29}$$

with $\partial\Omega_D \cup \partial\Omega_N = \partial\Omega$ and κ, $c : \Omega \to \mathbb{R}^+$ are positive-valued coefficients. We now work throught the problem in the usual way before applying hybridization. We rewrite the problem as a first order system to get the mixed problem:

$$\mu\mathbf{u} + \nabla p = 0, \text{ inside the domain } \Omega \tag{2.30}$$
$$\nabla \cdot \mathbf{u} + cp = f, \text{ inside the domain } \Omega \tag{2.31}$$
$$p = p_0 \text{ on } \partial\Omega_p \tag{2.32}$$
$$\mathbf{u} \cdot \mathbf{n} = g, \text{ on } \partial\Omega N \tag{2.33}$$

where $\mu = \kappa^{-1}$ and $\mathbf{u} = -\kappa\nabla p$ is the velocity variable.

Bringing this into the weak form we seek to find $(\mathbf{u}_h, p_h)$ in finite-dimensional subspaces $\mathbf{U}_h \times V_h \subset \mathbf{H}(div) \times L^2$ such that:

$$\sum_{K \in T_h} \int_K \mathbf{w} \cdot \mu \mathbf{u}_h dx - \sum_{K \in T_h} \int_K \nabla \cdot \mathbf{w} \cdot p_h dx = -\sum_{e \in \partial\Omega_D} \int_e \mathbf{w} \cdot \mathbf{n} \cdot p_0 ds \quad \forall \mathbf{w} \in \mathbf{U}_{h,0} \tag{2.34}$$
$$\sum_{K \in T_h} \int_K \phi \cdot \nabla \cdot \mathbf{u}_h dx + \sum_{K \in T_h} \int_K \phi \cdot cp_h dx = \sum_{K \in T_h} \int_K \phi \cdot f dx \quad \forall \phi \in V_h \tag{2.35}$$

with:

$$\mathbf{U}_h = \{\mathbf{w} \in \mathbf{H}(div; \Omega) : \mathbf{w}|_K \in \mathbf{U}(K), \forall K \in T_h, \mathbf{w} \cdot \mathbf{n} = g \text{ on } \partial\Omega_N\} \tag{2.36}$$
$$V_h = \{\phi \in L^2(\Omega) : \phi|_K \in V(K), \forall K \in T_h\} \tag{2.37}$$

and where $\mathbf{U}_{h,0} \subset \mathbf{U}_h$ is the space of vector-valued functions in $\mathbf{U}_h$ whose normal components vanish on $\partial\Omega_N$. Typical choices of $\mathbf{U}(K)$ include Raviart-Thomas (RT), Brezzi-Douglas-Marini (BDM) and Brezzi-Douglas-Fortin-Marini (BDFM) elements. [4]. Now we expand the solutions in terms of $\{\mathbf{\Psi}_i\}_i$, the basis for $\mathbf{U}_h$, and $\{\xi_i\}_i$, the basis for V_h:

$$\mathbf{u}_h = \sum_{i=1}^{N_u} U_i \mathbf{\Psi}_i, \quad p_h = \sum_{i=1}^{N_p} P_i \xi_i \tag{2.38}$$

with U_i and P_i being the coefficients to be determined. Letting $\mathbf{w} = \mathbf{\Psi}_j$, $j \in \{1, ..., N_u\}$ and $\phi = \xi_j$, $j \in \{1, ..., N_p\}$ in 2.34 - 2.35, we get the following discrete saddle point system:

$$\begin{bmatrix} A & -B^T \\ B & C \end{bmatrix} \begin{pmatrix} U \\ P \end{pmatrix} = \begin{pmatrix} F_0 \\ F_1 \end{pmatrix} \tag{2.39}$$

with $U = \{U_i\}, P = \{P_i\}$ the coefficient vectors and

$$A_{ij} = \sum_{K\in T_h} \int_K \mathbf{\Psi}_i \cdot \mu \mathbf{\Psi}_j dx \tag{2.40}$$

$$B_{ij} = \sum_{K\in T_h} \int_K \xi_i \cdot \nabla \cdot \mathbf{\Psi}_j dx \tag{2.41}$$

$$C_{ij} = \sum_{K\in T_h} \int_K \xi_i \cdot c\xi_j dx \tag{2.42}$$

$$F_{0,j} = -\sum_{e\in\partial\Omega_D} \int_e \mathbf{\Psi}_j \cdot \mathbf{n} \cdot p_0 ds \tag{2.43}$$

$$F_{1,j} = \sum_{K\in T_h} \int_K \xi_j \cdot f dx \tag{2.44}$$

Now we have worked through the problem and brought it into a form we can solve. There are many methods to solve such a system, but here we will look at a hybridized mixed method.

Hybridized mixed method example

1. To create the discontinuous variant of our problem, we replace the solution for $\mathbf{u}_h$ with the space U_h^d with
$$U_h^d = \{\mathbf{w} \in (L^2(\Omega))^n : \mathbf{w}|_K \in U(K), \forall K \in T_h\} \tag{2.45}$$
The differences with U_h, the original space, is that U_h^d is a subspace of $(L^2(\Omega))^n$ thus contains local $\mathbf{H}(div)$ functions. Additionally, normal componets are no longer continuous on ∂T_h, the set of facets of T_h.

2. Introduce Lagrange multipliers as an additional variable in the *space of approximate traces*, M_h:
$$M_h = \{\gamma \in L^2(\partial T_h) : \gamma|_e \in M(e), \forall e \in \partial T_h\} \tag{2.46}$$
with $M(e)$ denoting a polynomial space defined on each facet.

3. Take 2.30, test with $\mathbf{w} \in \mathbf{U}_h^d(K)$ and integrate by parts over each element K. Introduce the trace function λ_h in surface integrals to approximate p_h on elemental boundaries. This results in:
$$\int_K \mathbf{w} \cdot \mu \mathbf{u}_h^d dx - \int_K \nabla \cdot \mathbf{w} \cdot p_h dx + \sum_{e\in\partial K} \int_e \mathbf{w} \cdot \mathbf{n} \cdot \lambda_h ds = -\sum_{e\in\partial K\cap\partial\Omega_D} \int_e \mathbf{w} \cdot \mathbf{n} \cdot p_h ds \tag{2.47}$$

4. Add an extra constraint equation in order to close the system, yielding the *hybridizable* form: find $(\mathbf{u}_h, p_h, \lambda_h) \in \mathbf{U}_h^d \times V_h \times M_h$
$$\sum_{K\in T_h} \int_K \mathbf{w}\cdot\mu\mathbf{u}_h^d dx - \sum_{K\in T_h} \int_K \nabla\cdot\mathbf{w}\cdot p_h dx + \sum_{e\in\partial T_h\setminus\partial\Omega_D} \int_e [[\mathbf{w}]]\cdot\lambda_h ds = -\sum_{e\in\partial\Omega_D} \int_e \mathbf{w}\cdot\mathbf{n}\cdot p_0 ds, \quad \forall \mathbf{w} \in \mathbf{U}_h^d, \tag{2.48}$$
$$\sum_{K\in T_h} \int_K \phi \cdot \nabla \cdot \mathbf{u}_h^d dx + \sum_{K\in T_h} \int_K \phi \cdot c p_h dx = \sum_{K\in T_h} \int_K \phi \cdot f dx, \quad \forall \phi \in V_h, \tag{2.49}$$
$$\sum_{e\in\partial T_h\setminus\partial\Omega_D} \int_e \gamma \cdot [[\mathbf{u}_h^d]] ds = \sum_{e\in\partial\Omega_N} \int_e \gamma \cdot g ds, \quad \forall \gamma \in M_{h,0} \tag{2.50}$$
where $M_{h,0}$ is the space of traces vanishing on $\partial\Omega_D$. $\mathbf{u}_h^d$ will be equal to $\mathbf{u}_h$ if M_h is chosen such that the normal components of $\mathbf{w} \in \mathbf{U}_h$ are in the same polynomial space as the trace functions.

5. As before, we expand the function in terms of bases for $\mathbf{U}_h^d$, V_h and M_h to get the matrix system:
$$\begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix} \begin{pmatrix} U^d \\ P \\ A \end{pmatrix} = \begin{pmatrix} F_0 \\ F_1 \\ F_2 \end{pmatrix} \tag{2.51}$$

6. Since $\mathbf{U}_h^d$ and V_h are discontinuous, U^d and P are coupled within the cell only. Hence, we can use *local static condensation* to eliminate these unknowns simultaneously and thus produce a smaller *global hybridized problem* for the trace unknowns Λ:

$$S\Lambda = E \tag{2.52}$$

   where $S = \{S^K\}_{K\in\Omega_h}$ and $E = \{E^K\}_{K\in\Omega_h}$ which are assmbled by gathering local contributions. We can solve this system through purely local operations (see Gibson 2018 [4]).

7. Once system 2.52 is solved to find Λ, U^d and P can be recovered locally in each element (see Gibson 2018 [4]).

2.2.3 Static condensation

Static condensation, also called Guyan reduction, reduces the number of degrees of freedom by eliminating interior unknowns and thus leading to a system defined on cell-interfaces only. [4, 26]

2.2.4 Local post-processing

Local post-processing can produce improved conservation properties or *superconvergent approximations*. Superconvergent approximations are those that have one order of accuracy more than solutions which have not been locally post-processed [4].

2.3 FEniCS

There are many existing finite element solvers, each offering slightly different features (e.g. possible mesh elements, visualization, solver quadrature) and focused on varying purposes. Examples of these are deal.II [27], a massively-parallel general-purpose object-oriented finite element library, and FEBio [28], a finite element analysis package for biomechanics and biophysics.

Most notable in relation to Firedrake is FEniCS [29], a toolchain used to implement finite element methods to solve problems. It uses the finite element method to automatically generate finite element codes. These are computer codes of the algorithms coming out of FEM which calculate the approximate solution for the given differential equation.

The FEniCS environment has been created to allow the development of finite element solvers within Python. The novel addition from this project was an easy to use tensor contraction representation, which lowers the number of operations used for computing integrals using Gaussian quadrature. [3]

Many parts of FEniCS's toolchain are also a part of Firedrake's toolchain. In comparison to FEniCSs, however, Firedrake is designed to have an even more separated set of abstractions, making it easier for individuals to contribute their expertise. [5]

2.4 Firedrake

Firedrake [5] is a toolchain that serves as a compiler for a domain-specific langage (DSL). Given a partial differential equation (PDE) and a discretisation method, it uses finite elements to generate code to approximately solve the equation.

It does this by creating intermediate representations at many different levels of abstraction. This allows the decoupling of defining the equation and its discretisation from generating efficient code to perform the calculations. Each layer of abstraction provides the opportunity to apply different types of optimisations to ultimately yield more efficient code.

It is this separation of the FEM abstraction further than ever before that makes Firedrake different from previous finite element systems. [5] It not only splits up using the finite element method from implementing it, as FEniCS does, but it also separates the creation of cell-local kernels from computing them efficiently over all the cells in the mesh, using a new abstraction called PyOP2. This means that, unlike for other systems, contributors don't need to deeply understand the finite element method.

Where possible, Firedrake re-uses existing abstractions to avoid re-invention and make it easier for users and contributors who might be familiar with these abstractions. [5]

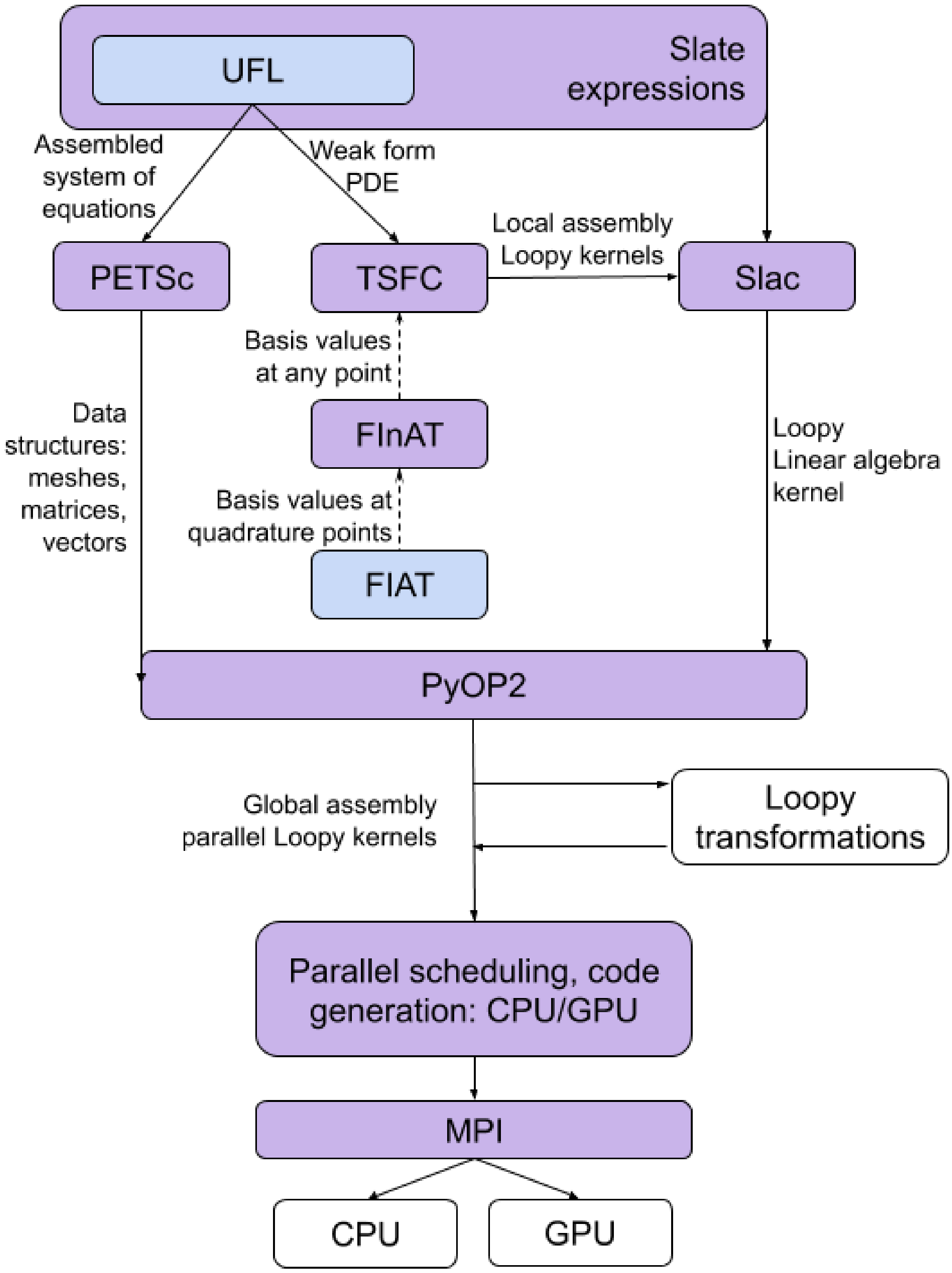

UFL
Slate expressions
Assembled system of equations
Weak form PDE
Local assembly Loopy kernels
PETSc
TSFC
Slac
Basis values at any point
FInAT
Basis values at quadrature points
FIAT
Data structures: meshes, matrices, vectors
Loopy Linear algebra kernel
PyOP2
Global assembly parallel Loopy kernels
Loopy transformations
Parallel scheduling, code generation: CPU/GPU
MPI
CPU
GPU

Figure 2.5: A basic diagram of the Firedrake toolchain. The abstractions originating from the FEniCS project are coloured blue. This is an updated and modified version of the original 2016 diagram found in [5]

Another feature that distinguishes Firedrake is its aim to be performance-portable i.e. have high performance on many different hardware platforms without needing to change its code. [3]

2.4.1 Firedrake architecture [5]

A basic diagram of the Firedrake toolchain is included in figure 2.5. The reader is strongly encouraged to follow through the diagram as we discuss some of the key abstractions below.

User input There are two main approaches to allowing the specification of the mathematics: stand-alone languages with their own grammar and syntax (e.g. freefem++ [30], GetDP [31]), or embedded languages (e.g. Unified Form Language [16], Sundance [32]) in existing general-purpose languages (in Python and C++ respectively).

Firedrake uses the Unified Form Language (UFL) [16] at its top layer. This is the same DSL as used in the FEniCS project. UFL is a DSL embedded in Python that allows users to express PDEs efficiently in a language close to mathematics. Firedrake's user interface is almost identical to the DOLFIN Python interface exposed by FEniCS. Firedrake models the higher-level mathematical objects using a combination of PyOP2 and PETSc objects.

TSFC [11] A *form compiler* takes in a definition of a weak form of PDEs and generates low-level code to compute it.

Until 2018, Firedrake used a modified version of the FEniCS Form Compiler (FFC) as its form compiler. This has now been replaced by the Two Stage Form Compiler (TSFC) [11].

FEniCS still uses FFC at this time, specifically with an implementation called the UFL Analyser and Compiler System (UFLACS).

Both UFLACS and TSFC take in variational forms defined in UFL and output low level code for the local assembly step. The output for the UFLACS is optimised C++ kernel functions. This would not fit in with Firedrake's PyOP2 layer, since PyOP2 expects to receive an unoptimised kernel that it then optimises for the given hardware. Hence TSFC generates unoptimised ASTs that represent the local assembly operations. [5]

The novel idea of TSFC is keeping the input expression at the highest abstraction for as long as possible such that optimisations can be done at this level. To make this process easier, the compilation done by the form compiler was split into two steps:

1. Change from the high-level form in UFL that includes finite element objects and geometry to the tensor algebra language *GEM* [11].
2. Generate abstract syntax trees for C to perform the tensor algebra expressions. These are optimized by later layers.

Keeping the structure of finite element forms is particularly important for discontinuous Galerkin problems.

FIAT The FInite element Automatic Tabulator (FIAT) [33] is a library responsible for producing finite elements. FIAT returns a *tabulation matrix* (just a numerical array) which has the values for the basis at each quadrature point.

FInAT FInAT [34] is a more abstract library of finite elements whose interface to form compilers allows for the implementation of optimisations crucial to generating sufficiently fast code for higher order finite element discretisations. The FInAT library's novel contribution is the interface it presents that makes it possible to rearrange finite element assembly loops such that one can express the structure within elements. Unlike FIAT, "FInAT Is not A Tabulator" and instead it allows users to evaluate basis functions at specific points (rather than just returning the values of the basis functions at the quadrature points like FIAT does). The expressions for these evaluations of the basis functions at arbitrary points are written in the tensor algebra language GEM [11], which is also the intermediate language used in TSFC.

FInAT serves as a generic FIAT wrapper, allowing TSFC to have access to all the functionality of FIAT through FInAT.

PyOP2 PyOP2 [5] is the main novel abstraction introduced by Firedrake. PyOP2 is a DSL embedded in Python that provides data structures for finite element operations, such as sparse matrices.

Recall that computational kernels (local operations) define the local assembly step of computing the integral of a test function against a trial function. PyOP2 allows users to define calculations that involve invoking these kernels for each entry in an unstructured mesh or fixed arity graph. It also allows users to specify their own kernels in either C or as an AST. PyOP2 then performs efficient parallel execution of these local kernels on each element in a mesh and assembles them into the global system of linear equations.

The introduction of PyOP2 in the system results in higher performance of mesh iteration across the board, as well as a much smaller code base leading to easier maintanence and extensibility. [5]

PETSc The results from local computations are assembled into a global system of linear equations that can be solved using exisiting libraries. Firedrake makes use of PETSc [5], a library with many solver algorithms.

MPI MPI [35] is a library which allows C or Fortran programs to communicate with each other. MPI routines are the means by which MPI processes (the C/Fortran program) communicate.

2.5 Slate and Slac [4]

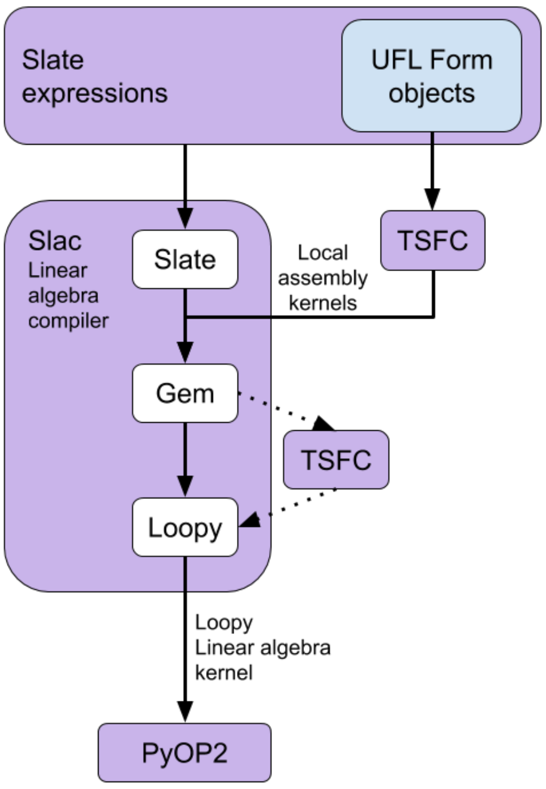


Figure 2.6: The Slate toolchain. The entries in white boxes are intermediate representations. This is a modified version of the original diagram found in [4]

In the local assembly step of FEM, the local contributions from each element in the mesh are found. Slate [4] is an embedded DSL Firedrake users use to express manipulations (linear algebra operations) on the finite element tensors coming from this stage.

UFL had a large influence on the design of Slate, especially in that both are compiled to low level code that executes the specified operations element-wise on the mesh. However, unlike for UFL, users of Slate can express complex linear algebra operations on the tensor objects.

Figure 2.6 shows the Slate toolchain. Slate expressions, which wrap UFL objects, are passed to the linear algebra compiler, Slac. In the newest development of Slac, it compiles these expressions in two steps: [36]

1. Using TSFC, translate Slate to Gem. Slac uses TSFC for its kernel functions for evaluating integral expressions.
2. Using TSFC, translate Gem to Loopy linear algebra kernels in C++ that will be applied cell-wise.

The final Loopy kernel is then passed to PyOP2, which wraps it in a mesh-iteration kernel. After the PyOP2 layer, parallelisation, code creation and compilation happen.

Important use cases of Slate are the automatic code generation for methods of hybridization, static condensation and localized post-processing. Hybridization and static condensation are implemented as runtime-configurable preconditioners via PETSc's Python interface. Slate can generate the code for local post-processing methods, since these can be expressed as local solves on each element. [4]

Slac/Slate is currently still using the Eigen library, however there is an ongoing migration from this to using the Loopy library. This project aimed to optimise for the new Loopy kernels.

2.6 Eigen [6]

Eigen is a widely used C++ template library for linear algebra. Eigen is different from other matrix libraries in that it is the only one that provides all of:

- it's fast
- it's versatile
  - handles all matrix types and sizes and all numeric types
  - provides various matrix decompositions and geometry features
  - has a rich ecosystem with many specialized modules
- it's reliable, with the reliability trade-offs of the algorithms available being well documented

It provides optimisations such as:

- Intelligent removal of temporaries
- Cache-friendliness for large matrices

Some additional advantages of Eigen include its small size, it being only a compile-time dependency, it being multi-platform and that compilation times stay reasonable when Eigen is used.

At the moment, Eigen only parallellizes general matrix-matrix products, hence parallel hardware is not used much by Eigen. [37]

2.7 Loopy [7, 8, 9]

Loopy is a loop generator that is embedded in Python. It allows the user to simply describe a computation and then the library transforms this code into code that will be highly performant on GPUs and multi-core CPUs. It targets array-type computations, such as those in PDE solvers like finite element. It targets the OpenCL/CUDA model of computation.

Loopy exposes the intermediate representations of the code to the user, so that they can be inspected and manipulated, allowing users control over the transformations applied. Users can implement their own transformations, but Loopy already provides many transformations such as:

- Change of data layout, e.g. switching between structure-of-arrays and array-of-structures memory layouts, padding, and block granularities.
- Loop unrolling and loop tiling.
- Prefetching and precomputation.
- Instruction-level parallelism.

Since the transformations performed are all done so explicitly by the user, the user takes on the responsibility of ensuring the observable behaviour of the code is unchanged. This has the benefit of allowing more different transformations to be applied as compared to those that conventional compiler architectures can safely perform.

Loopy transformations are controlled by a programming language, hence the code generated can be adapted for specific hardware or workloads.

Procedures in Loopy are called *Loopy kernels*.

2.8 Other Related Work

There are many systems for the efficient implementation of tensor algebra expressions, but many of these have little overlap of challenges resulting from their focus being on domains different from the finite element method. Such systems include libtensor [38], CTF [39] and TensorFlow [40]. Libtensor, for example, has been developed for quantum chemistry, where the tensors tend to be much larger than for those in finite element local assembly. [11]

There are also several projects that have tensor algebra languages in their intermediate stages. [41, 42, 43]

2.8.1 Dune-FEM and deal.II

An alternative way of implementing finite element solvers is using templated C++ code, as seen in systems such as DUNE-FEM [44] and deal.II [27]. A benefit of this method is that users have total control over all parts of the algorithm, hence this approach is more flexible and extendable.

However, users need to provide C++ implementations of the key kernels (such as the local operations), hence there is often a steep learning curve for new users and the user needs to have some experience in programming. This is in contrast to systems that use simple high-level languages for the user input, such as UFL used in Firedrake and FEniCS, which are good for prototyping but make it more difficult to extend the underlying C++ framework. Related is FreeFEM, which features a DSL embedded in C++ through the use of expression templates. [45]

In addition, for such templated C++ finite element solvers, the optimisations available are limited by the compiler available. In comparison, systems like Firedrake and FEniCS have domain-specific compilers that are able of optimising far beyond the limitations of general-purpose compilers. [46]

Vectorization in DUNE and Firedrake The vectorization strategy in Firedrake is automatically generated through the toolchain. This toolchain uses Loopy to apply sequences of transformations which support vectorization by grouping mesh entities such that each SIMD lane computes independently on one entity. [47]

Vectorization strategies in Dune for the assembly stages of Discontinuous Galerkin methods on hexahedral meshes make use of the tensor product structure of finite elements. The exact vectorization strategies applied to a given problem are chosen by a code generation toolchain that is integrated with dune-pdelab. This toolchain is very similar to that of Firedrake, including UFL combined with loopy. [45]

2.8.2 BLAS and LAPACK

LAPACK [48] (Linear Algebra PACKage) is a library for numerical linear algebra. It provides routines for solving linear systems of equations, eigenvalue problems, singular value problems, as well as implementing matrix factorizations, providing implementations in single and double precision for real and complex matrices alike. Much of the computation done within LAPACK routines is done by calling BLAS.

BLAS [49] (Basic Linear Algebra Subprograms) is a specification for low-level routines that perform linear algebra computations. There are many implementations of BLAS, many of which are optimised for particular machines.

BLAS implementations are often optimised for large tensors, while Eigen is efficient for small tensors [11]. Unlike Eigen, BLAS and LAPACK do not handle fixed-sized matrices nor sparse matrices. They also don't have as many convenience features, such as specialized modules for geometric operations. [37] Eigen speed is comparable to that of the best BLAS versions. [50]

In BLAS, the user must separate their desired operations into tiny steps that follow its fixed-function API. This leads to the creation of more temporaries. Eigen, on the other hand, can optimize operations globally, allowing it to be much faster than BLAS for computations that include complex expressions. [37]

2.8.3 EXCAFE [10]

Traditionally, the local assembly step in FEM has been performed using numerical quadrature. FFC and TSFC, present in FEniCS and Firedrake respectively, use tensor contraction as well as quadrature, depending on heuristics to choose the most suitable method for the given problem. The tensor contraction method does the local computation using the contraction of a reference tensor with an element-dependent geometry tensor [11].

EXCAFE [10] is a dense linear algebra active library that uses symbolic algebra techniques in an attempt to outperform these other implementations of local assembly. These techniques have been tried previously, for example in the FINGER [51] system and the SyFi Form Compiler [29] (a previous form compiler used in FEniCS).

EXCAFE delays evaluating library calls, storing this as part of a DAG, and then generates optimised code at runtime for the composition of library calls that have been delayed. The key optimisations it performs are loop fusions and contracting temporary arrays to scalars.

In EXCAFE local matrices are represented by the polynomial expressions that are used to evaluate the entries within them. This representation reduces the number of floating point operations required to evaluate the local matrix, and hence reduces the execution time required. The local assembly code generated in EXCAFE is often more efficient than that from tools that use the quadrature or tensor representations. [52]

All the parts of the EXCAFE toolchain are closely integrated, whereas the components in Firedrake are much more loosely coupled. This means EXCAFE is hard to weave in to existing toolchains, despite it being able to generate very efficient code. [3]

2.9 Summary

This concludes the outline of the mathematical concepts related to the finite element method, including some of the notation and depictions we will use throughout this thesis. This discussion includes Firedrake, its components and some of its supporting libraries.

The ideas in this chapter form the basis of our discussion about Firedrake, and will continue to contextualise the mathematics and the code which we produce in this project.

Chapter 3

Part 1: Reducing Unneccessary Temporaries

Through intelligent indexing, we were able to reduce the number of temporaries created from Slate operations. This resulted in the reduction of the amount of memory needed to store the temporaries. The implementation involved the modification of directed acyclic graphs (DAGs) composed of GEM nodes that represent operations on tensors.

We begin by briefly defining relevant terminology for UFL, GEM and Slate/Slac.

3.1 GEM and UFL terminology [11]

This section draws from the examples and definitions from Homoloya 2018 [11].
GEM and UFL distinguish between:

- *shape*: ordered list of the extent in each dimension.
- *free indices*: unordered set of "named" dimensions.

A good example of this taken from Homolya 2018 [11] is summarised in table 3.1.

Object	Shape	Free indices
B, a 2x2 matrix	(2,2)	none
$B_{(1,1)}$	(1,1)	none
$B_{(i,j)}$	(1,1)	i and j
$B_{(i,1)}$	(1,1)	i

Table 3.1: Examples of shape and free indices.

Throughout this thesis, we will be making use of the following subset of UFL nodes:

- **Indexed**(e, $(\alpha_1, \alpha_2, ...\alpha_r)$) (frequently written as $A[\alpha_1, ..., \alpha_r]$)
  indexes into expression e of rank r using multi-index α. The multi-index can be made up of fixed and free indices. The resulting object has scalar shape and the same free indices as in α.

  For example, let e be a tensor of shape (3, 4, 5) with no free indices. Then `Indexed(e, (1, j, k))` would have shape (1, 1, 1) and free indices j and k, j=1...4, k=1...5.
- **ComponentTensor**(e, $(\alpha_1, \alpha_2, ...\alpha_k)$)
  turns scalar expression e's free indices $\alpha_1, \alpha_2, ...\alpha_k$ into shape.

GEM and UFL as quite similar; the main difference between them is how they treat free indices. In UFL, free indices have the same constraints as shape, e.g. for addition the two tensors must have the same shape and the same free indices. GEM has the same constraints for shape, but it allows the result of an expression to be index-dependent. For example if u has shape (3,1), then $u_i + u_j$ is illegal in UFL but in GEM is a 3x3 matrix.

Below is a summary of the GEM nodes relevant to this project:

- Terminals: Variable
- Scalar operations: Delta (i.e. δ_{ij}=1 if $i = j$, else 0)
- Index types
  - Index (free index): results in a loop at code generation
  - FlexiblyIndexed
- Tensor nodes: Indexed, ComponentTensor (as for UFL)

3.2 Slate/Slac terminology [4, 11]

Slate is designed around UFL. However, unlike for UFL, users of Slate can express complex linear algebra operations on the tensor objects.

Note that in this thesis we are using the notation $f(x, y, z; \alpha, \beta)$ to denotes that the form f must be linear in its *coefficients* (x, y and z), but it may not necessarily be linear in its *arguments* (α and β).

Slate is made up of two main abstraction types: (1) 'terminal' tensors and (2) operations on terminal tensors.

Terminal tensors, also called *leaf nodes* or *terminals*, are nodes of the expression graph that have no children. Terminals associate a tensor with data on an element. This data is either in the form of a multi-linear integral or assembled data (i.e. a vector of coefficients). [4, 11]

- **Tensor($a(\mathbf{c}; \mathbf{v})$)**
  associates form $a(\mathbf{c}; \mathbf{v})$ with its local tensor, i.e.

$$A^k \leftarrow a(\mathbf{c}; \mathbf{v})|_K$$

  for all elements K of the mesh.

- **AssembledVector(f)**
  associates function f with its local coefficient vectors.

Operations on terminal tensors currently supported by Slate include: [4]

- **-A**, the additive inverse of A
- **Transpose(A)**
- **A.inv**, the inverse of A, with A a square tensor
- **A + B**, addition of A and B, with A and B equally shaped
- **A * B**, a contraction over the last index of A and the first index of B. This is simply multiplication for matrices, vectors and scalars.
- **A.solve(B, decomposition="...")**, the X obtained from solving $AX = B$. One can optionally specify a direct factorisation strategy.
- **A.blocks[indices]**
  where A is a tensor from a mixed finite elemnt space. This function return the block of A corresponding to the indices. For example, if A is

$$A = \begin{bmatrix} A_{00} & A_{01} \dots & A_{0m} \\ A_{10} & A_{11} \dots & A_{1m} \\ \vdots & \ddots & \vdots \\ A_{n0} & \dots & A_{nm} \end{bmatrix}$$

  Then

$$A_{\mathbf{pq}} = \begin{bmatrix} A_{p_1 q_1} & \dots & A_{p_1 q_c} \\ \vdots & \ddots & \vdots \\ A_{p_r q_1} & \dots & A_{p_r q_c} \end{bmatrix} \leftarrow \text{Block}(A, (\mathbf{p}, \mathbf{q}))$$

3.3 First steps

We begin by trying to understand the GEM expression DAGs produced for simple Slate operations, as more complex operations are built out of these simpler ones.

We predict that optimising these building-block operations would translate to more optimised complex expressions. This hypothesis forms a basis for our optimisation.

To help with understanding, we experimented with existing functions, manually inspected the code produced and visualised the structure of the GEM DAGs in the GEM to Loopy stage of compilation (as this is the stage during which we will have our unneccessary temporary removing step). We used Graphviz [53] to create a way to easily visualise these DAGs, both in a DAG-aware way to see what computations were being performed and also in a tree view to understand what part of each expression creates what nodes in the DAG. Several of the graphs created from this are included in this thesis.

3.4 Model problem

We chose our first model problem to be the discontinous Helmholtz equation on cell integrals. The code for the setup is included in 4.1. This means that for the below operations, we use

$$\begin{aligned} a &= \int \nabla v \cdot \nabla u + vu \, dx \\ L &= \int fv \, dx \\ f &= \left(1 + 8\pi^2\right) \cos\left(2\pi x\right) \cos\left(2\pi y\right) \end{aligned} \tag{3.1}$$

3.4.1 Tensor

We begin by considering the simplest example: creating a Tensor object. We will do this for object `a`, where `a` is as in 3.1. For this, we actually envoke `assemble(Tensor(a))`, since we need to call `assemble()` to evaluate `Tensor(a)`.

The original expression tree is given in figure 3.1. In our trees, `root` is an arbitrary name used by the ploter to represent the highest level of the expressions in order to support lists of expressions.

This originally produced the code:

```
for i1, i0
    t0[i0, i1] = T0[i0, i1]  {id=insn, priority=2}
end i1, i0
for i_0, i
    output[i, i_0] = output[i, i_0] + t0[i, i_0]  {id=insn_0, priority=1}
end i_0, i
```

Listing 3.1: Original code for `Tensor(a)`

Note the different forms for the possible names for a temporary:

1. Tn, where $n \in \mathbb{R}$, is the name of the temporary that stores the tensor when it is created. This is always a neccessary temporary.

2. tn, where $n \in \mathbb{R}$, is the name of an intermediate temporary, used to store intermediate results in the computation of an operation on a tensor.

In the code outputs, `output` denotes the buffer that we fill up at the end of our set of operations and it is this buffer that is passed on to other parts of the code.

In 3.1, T0 and t0 are the names of the temporaries created here. T0 is necessary as it stores the tensor object. t0 is created as an intermediate temporary used to perform the operation of Tensor(). We see that t0 is an unneccessary temporary.

After appling our optimisation, the resulting GEM expression DAG is as in figure 3.2. We see that the difference between the DAG before and the DAG after is the removal of the Component-Tensor node and its associated Indexed node.

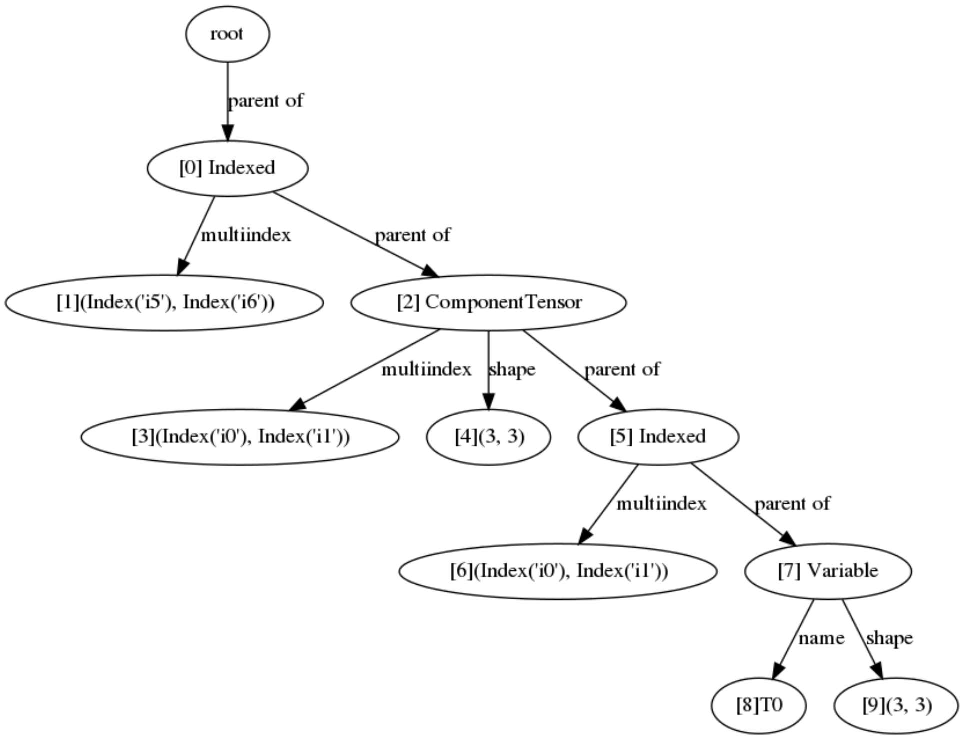


Figure 3.1: Tree representation of the GEM DAG created for operation `Tensor(a)`, before the optimisation.

Through this change, the underlying tensor can be more directly accessed. This leads to greatly simplified code, as we see in 3.2. In this code, there is only one temporary used. This is of the form Tn, hence we know it is the necessarily one which holds the actual tensor. From this we see that the resulting code is the optimal one for this operation - there are no unneccessary temporaries.

```
for i_0, i
    output[i, i_0] = output[i, i_0] + T0[i, i_0]  {id=insn, priority=1}
end i_0, i
```

Listing 3.2: Optimised code for `Tensor(a)`

3.4.2 Transpose

Next, we look at the example of `Tranpose(Tensor(a))`, where is `a` as in 3.1. For this, we actually envoke `assemble(Transpose(Tensor(a)))`.

The original expression tree is given in figure 3.3.

We can express the transpose operation in terms of the GEM/UFL operations as

$$\text{Tranpose: ComponentTensor}(A[i,j],(j,i)) \tag{3.2}$$

The original code output for `Tranpose(Tensor(a))` is given in 3.3 below.

```
for i1, i0
    t0[i0, i1] = T0[i0, i1]  {id=insn, priority=3}
end i1, i0
for i6, i5
    t1[i6, i5] = t0[i5, i6]  {id=insn_0, priority=2}
end i6, i5
for i_0, i
    output[i, i_0] = output[i, i_0] + t1[i, i_0]  {id=insn_1, priority=1}
end i_0, i
```

Listing 3.3: Original code for `Transpose(Tensor(a))`

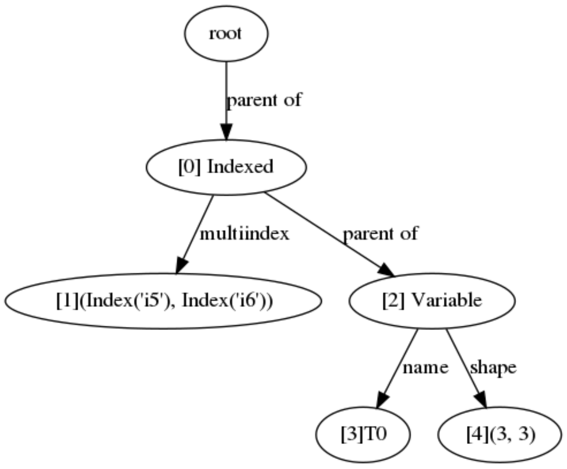


Figure 3.2: Tree representation of the GEM DAG created for operation `Tensor(a)`, after the optimisation.

In 3.3, we see we have three temporaries created, T0, t0 and t1. T0 is necessary as it stores the tensor object. t0 and t1 are intermediate temporaries used to perform the operation of Transpose(Tensor()). We see that t0 and t1 are unneccessary temporaries.

The same operation can be performed by just flipping the indices at which we are accessing T0. This then means we don't need to create t0 and t1. As before, the creation of one temporary, T0, is unavoidable.

We achieved this smarter indexing and thus removal of the need to create t0 by removing the ComponentTensor and its associated nodes from the GEM DAG. The tree representation of the resulting GEM expression is given in figure 3.4 and the generated code is given in 3.4.

```
for i, i_0
    output[i_0, i] = output[i_0, i] + T0[i, i_0]  {id=insn, priority=1}
end i, i_0
```

Listing 3.4: Optimised code for `Transpose(Tensor(a))`

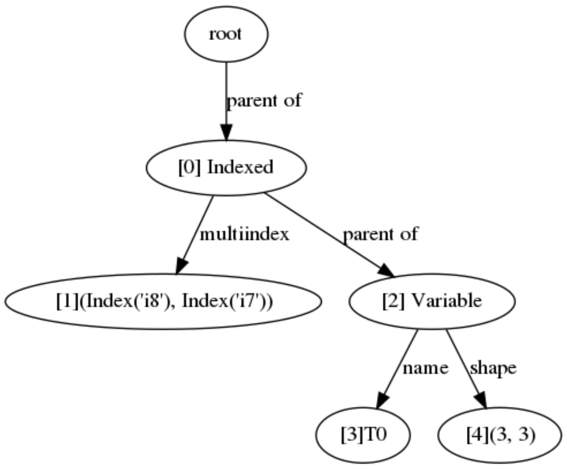


Figure 3.4: Tree representation of optimised GEM DAG created from `Transpose(Tensor(a))`.

The benefit of this is more pronounced when considering stacking operations, for example for `Transpose(Transpose(Tensor(a)))` we reduced the number of temporaries from 5 to just 1.

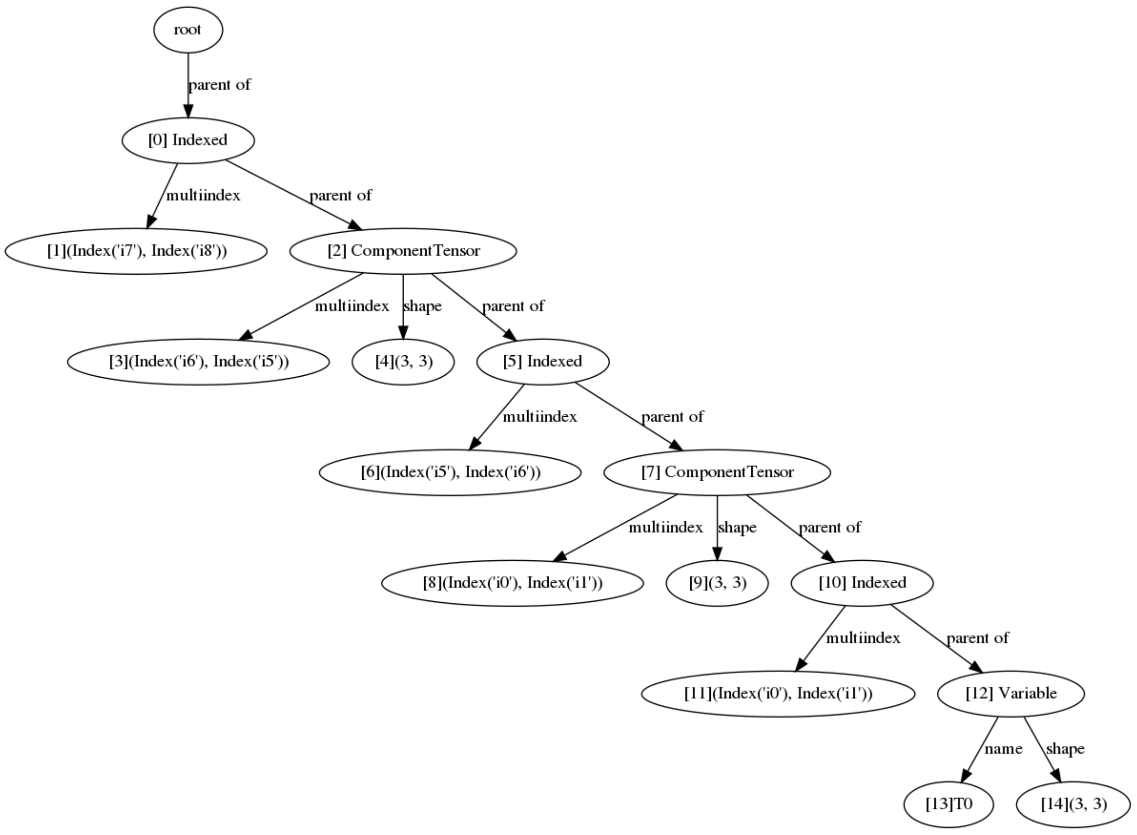


Figure 3.3: Tree representation of original GEM DAG created from `Transpose(Tensor(a))`.

3.4.3 Other basic Slate operations

We include the code generated before and after removing these ComponentTensor nodes for the remainder of the basic Slate operations in the appendix. For all these operations we see the same pattern of fewer temporaries needed, details in table 4.1, as well as shorter, more succinct code being generated that continued to have the same behaviour.

3.5 Implementation

How this optimisation is actually implemented in Firedrake is by passing through the GEM expression DAG before code generation and *eliminating the ComponentTensor nodes.*

As discussed in Homoloya 2018, this transformation can be understood as removing non-scalar shapes in non-terminal nodes using the rule: [11]

$$\text{Indexed}(\text{ComponentTensor}(e, \alpha), \beta) \to e|_{\alpha \to \beta} \tag{3.3}$$

with e being a scalar-shaped expression, α a multi-index made up of free indices of e and β a multi-index with the same rank as α. $e|_{\alpha\to\beta}$ denotes of substitution of indices from α_i to β_i $\forall i$.

3.6 Relation to existing work

This elimination of ComponentTensors is the same transformation that is performed in the second stage of TSFC, the translation of GEM to C. In TSFC this transformation is done to make the code generation easier. [11]

3.7 Summary

In this chapter we discussed the implementation of our optimisation to reduce the number of temporaries created when performing operations on tensors. This optimisation involves passing through the GEM expression DAG before code generation and eliminating the `ComponentTensor` nodes.

Through looking at examples of expression trees and output code, we demonstrated that our optimisation results in the removal of `ComponentTensors` in the DAGs and leads to the reduction of the number of temporaries in the code.

In making this optimisation, our goal was to reduce the memory needed to perform the sequence of operations. We therefore want to investigate the specific effects that a reduction in temporaries, as we saw in this chapter, has in reducing the memory overhead.

Chapter 4

Part 1: Reducing Unneccessary Temporaries: Evaluation and Results

4.1 Aims

This project aims to optimise the local assembly layer in Firedrake. In this part of the project, we focused on trying to remove the *number of temporaries created*, which we achieved through modifying and removing nodes in the GEM expression DAG.

We measure success by the reduction of the *working set size* (WSS), i.e. the amount of memory needed to perform the sequence of operations. Reducing the working set size allows more information to fit into faster memory and thus can also reduce the *calculation time*. We care about the WSS only for the resulting code, not for the compiler itself, hence we don't really care about the effect of the optimisations on the size of the GEM DAG itself, for example.

We approximate the working set size of the compiler from the resulting source code, in particular by looking at the amount of memory taken up by the temporaries.

Importantly, we must ensure that the behaviour of Slate does not change. We need to check that the generated code after optimisation implementation gives us an error not larger than after our optimisations have been implemented.

4.2 Analysis of the optimisation using modelling

We investigate various operations to determine a model for the change in working set size. We perform experiments before and after the optimisations have been applied.

The experiments are done on a "typical" laptop (exact specification provided in the appendix).

We begin by investigating simple, minimal examples, as seen in the previous chapter and then look into larger, more involved examples to see how working set size scales. We want to particularly focus on common and significant types of problems.

An important use case for Slate is hybridization so that methods such as static condensation and local post-processing can be employed, leading to systems that can be solved more easily. Therefore we use our model to predict the memory taken by temporaries for a hybridization example, see 4.2.6.

We start by figuring out how much memory in bytes is taken up by temporaries in a loopy kernel. This is as below:

$$\text{memory size of temporaries} = \sum_{t \in \text{temporaries}} \prod_{s \in \text{shape of t}} s * \text{data type size} \tag{4.1}$$

The data type in our case is always float64 so will be 8 bytes. The only things that can change for us are the shape of the temporaries and the number of temporaries.

In figure 4.1 we include a diagram showing the relationship between the key objects that can affect the shape of temporaries as implemented in Firedrake. The effects of these will be discussed in more detail in later parts of this chapter.

Note the implementation of `FiniteElement` has the same structure as Ciarlet's definition 2.1.3: `cell` here is the domain of the finite element (related to K); `degree` is the maximum degree of the

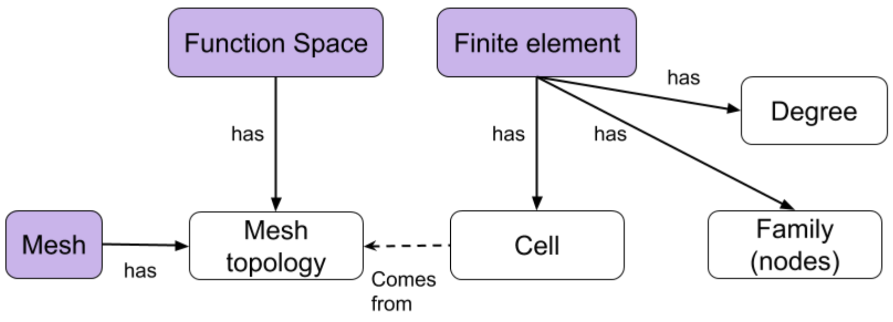


Figure 4.1: Firedrake implementation of `FunctionSpace`, `FiniteElement` and `Mesh`.

polynomials in the space (related to P); `family` is the finite element family, i.e. the type of finite element (related to N). The full list of elements supported in Firedrake can be found at [54].

We begin by analysing the below example (same as model problem 3.1 in chapter 3):

```
# Discontinuous Helmholtz equation on cell integrals
mesh = UnitSquareMesh(5, 5)
V = FunctionSpace(mesh, "DG", 1)
u = TrialFunction(V)
v = TestFunction(V)
f = Function(V)
x, y = SpatialCoordinate(mesh)
f.interpolate((1+8*pi*pi)*cos(x*pi*2)*cos(y*pi*2))
a = (dot(grad(v), grad(u)) + v * u) * dx
L = f * v * dx
```

Listing 4.1: UFL for setting up discontinuous Helmholtz on cell integrals

We get the following kind of information about the temporaries in the loopy kernel.

```
T0: type: np:dtype('float64'), shape: (10, 10), dim_tags: (N1:stride:10, N0:stride:1)
    scope:local
t0: type: np:dtype('float64'), shape: (10, 10), dim_tags: (N1:stride:10, N0:stride:1)
    scope:local
t2: type: np:dtype('float64'), shape: () scope:local
t3: type: np:dtype('float64'), shape: (10), dim_tags: (N0:stride:1) scope:local
```

Listing 4.2: Example of the information given about temporaries in Loopy kernel.

4.2.1 Operation type

We hypothesise that the types of operations determine the number of temporaries. We also hypothesise that our model can accurately predict the number of temporaries required for a sequence of operations from the temporaries needed for each basic operation that needs to be performed as part of that set of operations. This relates to our earlier hypothesis, 3.3, where we predict that by reducing the memory for temporaries in the basic building-block operations, we will also reduce the memory for more complex expressions.

Operation table We include in table 4.1 below the number and shape of temporaries needed for basic Slate operations, both as individual operations and when stacked with other operations, before and after optimisation.

Here we introduce a new notation that we will using in this thesis: a notation for the number of temporaries of a certain shape. We let `n` be the number of temporaries of the given shape, and the second part denote the shape of temporaries, as below:

- `n (.,.)` where `(.,.)` denotes a matrix temporary with a shape that is a 2-tuple;
- `n (.)` where `(.)` denotes a vector temporary with a 1-tuple shape;
- `n ()` where `()` denotes a scalar quantity i.e. just a number.

In the table, with a and L as in 4.1

$$u = Function(V) \tag{4.2}$$
$$A = Tensor(a) \tag{4.3}$$
$$F = AssembledVector(assemble(L)) \tag{4.4}$$
$$b = AssembledVector(Function(assemble(L))) \tag{4.5}$$

The expressions in square brackets are a simple representation of the operation intended to aid understanding, these are not full, valid UFL expressions (for example, since they do not call assemble()).

The stacked before/after columns represent the number of additional temporaries that are needed when that operation is added to a sequence of operations. This can be used to account for the *stacking effect* in which some temporaries are re-used by the compiler from previous optimisations. Its important to note that such stacking results in a smaller number of temporaries, and sometimes in different shaped temporaries, than would be expected by treating the operations in isolation.

Operation	Temps before	Stacked before	Temps after	Stacked after [1]
Tensor	2 (.,.)	2 (.,.)	1 (.,.)	$1(.,.)^2$
AssembledVector	1 (.)	1 (.)	0	0
Negative [-A]	3 (.,.)	1 (.,.)	1 (.,.)	variable [3]
Transpose [A.T]	3 (.,.)	1 (.,.)	1 (.,.)	0
Inverse local [A.inv]	3 (.,.)	1 (.,.)	2 (.,.)	1 (.,.)
Addition of matrices [A+A]	3 (.,.)	1 (.,.)	1 (.,.)	0
Addition of vectors [b+b]	3 (.,.)	1 (.)	1 (.,.)	0
Multiplication Matrix-Vector	5: 2 (.,.) + 3 (.)	2 (.)	1 (.,.)	variable [3]
Multiplication Matrix-Matrix	6 (.,.)	2 (.,.)	2 (.,.)	variable [3]
Global Solve [solve(A, u, F)]	5: 2 (.,.) + 3 (.)	1 (.)	1 (.,.)	0
Local solve [A.solve(F)]	6: 3 (.,.) + 3 (.)	1 (.,.) + 1 (.)	2 (.,.)	1 (.,.) + 1 (.)
Blocks	0	0	0	0

Table 4.1: The number and shape of temporaries needed for basic Slate operations, both as individual operations and when stacked with other operations, before and after optimisation.

[1] N.b. The last operation that needs to be performed can be done so directly outputting into the output buffer, hence doesn't need a temporary.
[2] This is of the form Tn with n a number. This is a necessary temporary.
[3] Some operations have several possible numbers of temporaries depending on the stacking context of the operation. These operations are discussed in the following section.

In table 4.2 we include some simple examples of stacked operations to demonstrate the stacking effect.

Operation	Temps before	Temps after
Negative stacked [-(-A)]	4 (.,.)	1 (.,.)
Transpose stacked [(A.T).T]	4 (.,.)	1 (.,.)
Addition stacked [A+(A+A)]	4 (.,.)	1 (.,.)
-A.inv.T	5 (.,.)	2 (.,.)
(-A.T + A) * b	8: 5 (.,.) + 3 (.)	1 (.,.)

Table 4.2: Table of the number and shapes of temporaries for several simple stacked operations.

Context-specific temporaries

The number and shape of temporaries post-optimisation for some operations is context-dependent. We outline these below, taking `A`, `B`, `F`, and `G` as defined in 4.14.

Negation The number and size of temporaries can be summarised as follows:

```
if negation before a matrix-matrix multiplication:
    if only 1 negation:
        then that negation adds 1 ()
    if > 1 negation:
        1 () temporary needed for the negation absorbed into the multiplication
        + 1 (.,.) for all other negations (i.e. num negation - 1 of (.,.))
elif negation on the vector in a matrix-vector multiplication:
    1 (.) for any number of negations on the vector
else:
    0 new temporaries
```

Listing 4.3: Algorithm for determining the number and size of temporaries for stacked negation post-optimisation

For some illustrative examples:

- **No mutiplications**: `-A.inv.T*F` has 1 temporary needed for the inverse only, no temporaries needed for negation.

- **Mat-mat multiplication, one negation**: `-A*A` needs 1 () temporary for negation, t0, and 1 (.,.), T0, for storing A. As we see with the output below, the negation is absorbed into the loop for the multiplication, thus yielding a temporary of shape ().

```
for i_0, i
    t0 = (-1.0)*T0[i, i_0]  {id=insn, priority=2}
    for i_1
      output[i, i_1] = output[i, i_1] + t0*T0[i_0, i_1]  {id=insn_0, priority=1}
end i_0, i, i_1
```

Listing 4.4: Optimised output for `-A*A`

- **Mat-mat multiplication, multiple negations**: `-A*-A*-B` includes two multiplications and three negations. We see one multiplication is performed on line 13, resulting in 1 (.,.) temporary, t3. The other multiplication is performed on the last line with the output heading straight to the output buffer, hence creates no temporaries. One of the negations is 'absorbed' into one of the multiplications, resulting in 1 () temporary, t2. The other 2 negations are performed in lines 2 and 5, with each one resulting in 1 (.,.) temporary.

```
1  for i, i_0
2      t0[i, i_0] = (-1.0)*T1[i, i_0]  {id=insn, priority=6}
3  end i, i_0
4  for i_2, i_1
5      t1[i_1, i_2] = (-1.0)*T0[i_1, i_2]  {id=insn_0, priority=5}
6  end i_2, i_1
7  for i_3, i_4
8      t3[i_4] = 0.0  {id=insn_1, priority=4}
9    end i_4
10   for i_5
11     t2 = (-1.0)*T1[i_3, i_5]  {id=insn_2, priority=3}
12     for i_6
13       t3[i_6] = t3[i_6] + t2*t0[i_5, i_6]  {id=insn_3, priority=2}
14   end i_5, i_6
15   for i_7, i_8
16       output[i_3, i_8] = output[i_3, i_8] + t3[i_7]*t1[i_7, i_8]  {id=insn_4,
       priority=1}
17 end i_3, i_7, i_8
```

Listing 4.5: Optimised output for `-A*-A*-B`

By comparing this code to that of `-B*-A*-A*-B` we see that despite already finding `-B` and saving this in temporary t0 in line 2, we again find `-B` and save it to a different temporary t1 in line 5, thus creating unneccessary temporaries. This means that no matter what the negation is in front of, even if it has been negated before, it will be follow the rules for the number of temporaries mentioned in the summary above.

Secondly, we see that only one multiplication "absorbs" one negation. So no matter how many multiplications there are following the negation, the rules outlined above still apply.

```
for i, i_0
    t0[i, i_0] = (-1.0)*T1[i, i_0]  {id=insn, priority=9}
end i, i_0
for i_1, i_2
    t1[i_1, i_2] = (-1.0)*T1[i_1, i_2]  {id=insn_0, priority=8}
end i_1, i_2
for i_3, i_4
    t2[i_3, i_4] = (-1.0)*T0[i_3, i_4]  {id=insn_1, priority=7}
end i_3, i_4
for i_5, i_6
    t4[i_6] = 0.0  {id=insn_2, priority=6}
  end i_6
  for i_7
    t3 = (-1.0)*T0[i_5, i_7]  {id=insn_3, priority=5}
    for i_8
      t4[i_8] = t4[i_8] + t3*t0[i_7, i_8]  {id=insn_4, priority=4}
  end i_7, i_8
  for i_9
    t5[i_9] = 0.0  {id=insn_5, priority=3}
  end i_9
  for i_11, i_10
      t5[i_11] = t5[i_11] + t4[i_10]*t1[i_10, i_11]  {id=insn_6, priority=2}
  end i_11, i_10
  for i_12, i_13
      output[i_5, i_13] = output[i_5, i_13] + t5[i_12]*t2[i_12, i_13]  {id=insn_7,
    priority=1}
end i_5, i_12, i_13
```

Listing 4.6: Optimised output for `-B*-A*-A*-B`

- **Mat-vec multiplication, any number of negations**: `A*-(-F)` results in the introduction of temporary t0 of shape (.) which is used to negate the vector F.

```
for i
  t0[i] = (-1.0)*(-1.0)*VecTemp0[i]  {id=insn, priority=2}
end i
for i_0, i_1
    output[i_0] = output[i_0] + T0[i_0, i_1]*t0[i_1]  {id=insn_0, priority=1}
end i_0, i_1
```

Listing 4.7: Optimised output for `A*-(-F)`

Matrix-Matrix multiplication The number and size of temporaries can be summarised as follows:

```
if mat-mat multiplication before an inverse OR includes a tranpose and not the final
    operation:
        1 (.,.)
else:
    1 (.)
```

Listing 4.8: Algorithm for determining the number and size of temporaries for stacked matrix-matrix multiplication post-optimisation

- **Mat-mat multiplication, followed by inverse**: `(A*B).inv` uses 1 (.,.) for the inverse, and since the inverse operation works on square matrices only, the input temporary into it, i.e. the temporary output from `(A*B)` needs to be a matrix, i.e. of shape (.,.).

- **Mat-mat multiplication, not before inverse and no transposes**: `(B*A)*G`, with `B`, `A` matrices and `G` a vector, uses 1 (.), t0, for the matrix-matrix multiplication, as we see in 4.9

```
for i, i_0
    t0[i_0] = 0.0  {id=insn, priority=3}
  end i_0
  for i_2, i_1
      t0[i_2] = t0[i_2] + T1[i, i_1]*T0[i_1, i_2]  {id=insn_0, priority=2}
  end i_2, i_1
  for i_3
    output[i] = output[i] + t0[i_3]*VecTemp0[i_3]  {id=insn_1, priority=1}
end i, i_3
```

Listing 4.9: Optimised output for `(B*A)*G`

Matrix-Vector multiplication The number and size of temporaries can be summarised as follows:

```
if mat-vec mult includes an odd number of tranposes on the matrix OR any number of
    negations on the vector:
    1 (.)
else:
    1 ()
```

Listing 4.10: Algorithm for determining the number and size of temporaries for stacked matrix-vector multiplication post-optimisation

- **Odd number of transposes**: `B.T.T.T*G + F` and `B.T*G + F` have the same output code. The +F is included here just so that the last operation is not the operation of interest, i.e. the multiplication, since if it was then no temporary would be created for it. The reason the code is the same is because the ultimate operations to be performed are the same, e.g. 3 transposes are the same as 1 transpose and the compiler is smart enough to just do 1 tranpose rather than 3. An odd number of tranposes one after another results in just one tranpose operation needing to be done.

  The line of code responsible for the matrix-vector multiplication between matrix B tranposed some number of times and vector G is also the one in which the transpose of B happens.

```
for i
  t0[i] = 0.0  {id=insn, priority=3}
end i
for i_0, i_1
    t0[i_1] = t0[i_1] + T0[i_0, i_1]*VecTemp1[i_0]  {id=insn_0, priority=2}
end i_0, i_1
for i_2
  output[i_2] = output[i_2] + t0[i_2] + VecTemp0[i_2]  {id=insn_1, priority=1}
end i_2
```

Listing 4.11: Optimised output for `B.T*G + F`. Same code as for `B.T.T.T*G`

- **Even number of transposes** For comparison, when we consider `B.T.T*G + F`, or this with any even number of tranposes on the matrix `B` we find that, though similar logic as above, no transposes need to be done. For these operations, we get the code below, where t0 is a temporary of shape ().

```
for i
  t0 = 0.0  {id=insn, priority=3}
  for i_0
    t0 = t0 + T0[i, i_0]*VecTemp1[i_0]  {id=insn_0, priority=2}
  end i_0
  output[i] = output[i] + t0 + VecTemp0[i]  {id=insn_1, priority=1}
end i
```

Listing 4.12: Optimised output for `B*G + F`, `B.T.T*G + F`, `B.T.T.T.T*G + F`, etc.

Interestingly, when we compare the codes for the odd 4.11 and even 4.12 cases, it raises the question of why the temporary in the odd case has the larger size of (.). We provide the following explanation: that for 4.11, because of the need to perform the tranpose operation on the matrix, the code generated has been split into separate loops for all parts of the process, whereas for 4.12 these loops have all been nested thus leading to a smaller temporary needed. Since the tranpose happens within the same instruction as the multiplication with the vector, there is no need for the for loops to have been separated. Hence the code we would want here would look something like 4.13 below, with t0 a () shaped temporary.

```
for i
  t0 = 0.0  {id=insn, priority=3}
  for i_0
    t0 = t0 + T0[i_0, i]*VecTemp1[i_0]  {id=insn_0, priority=2}
  end i_0
  output[i] = output[i] + t0 + VecTemp0[i]  {id=insn_1, priority=1}
end i
```

Listing 4.13: The code we wish to be outputed for `B.T*G + F`

From this we see that our optimisation has not reduced the memory needed by temporaries to the optimal level and there are still improvements that can be done with respect to this. The improvement for this example does not seem very significant, so may not be worth pursuing unless we come across an example that performs many transposes that needs to minimise the working memory as much as possible.

4.2.2 Temporary type

The *size of the shape tuple* depends on the type of tensor being stored in the temporary. For matrices, temporaries have a 2-tuple for shape, while `AssembledVectors`, which are one dimensional vectors, have a 1-tuple for shape. There are also temporaries that have a shape of (), which take up just 8 bytes of memory and correspond to just a scalar number, usually 0, and can be seen in some cases of multiplications.

4.2.3 Finite element family

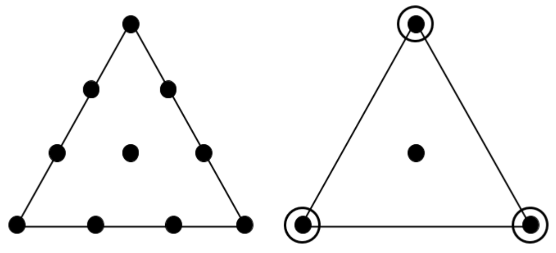
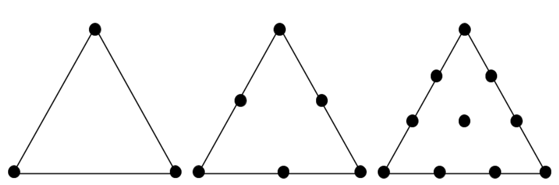


Figure 4.2: Showing the effect of having a different finite element family. Left is Lagrange; right is Hermite. If K and P match, as they do here, then the number of degrees will be the same. We note that finite element families differ by N and by P.

The *values in the shape tuple* of the temporaries, i.e. the extent of the temporary in each dimension, is in each dimension equal to the degrees of freedom on a cell.

P is a polynomial space, which may not necessarily be all the polynomials of a particular degree. Finite element families can differ by the nodes N and also by the space P. [55]

If K and P match, the number of degrees of freedom will be the same. This is because the number of the degrees of freedom is equal to the number of entries in the nodal basis, which is a basis for P. It is known that, for a given vector space, all bases for the space have the same number of elements.


(a) Lagrange elements on triangle. From left to right, the degree is 1, 2, 3.

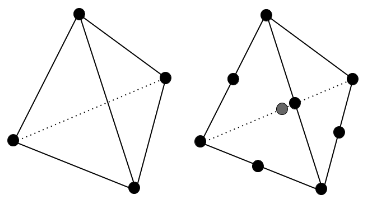

(b) Lagrange elements on tetrahedra. From left to right, the degree is 1, 2.

Figure 4.3: Lagrange elements on simplices.

4.2.4 Mesh and degree of finite element

The degrees of freedom on a cell, and hence the extent of the temporary in each dimension, depends on:

1. The cell shape i.e. **triangle vs quadrilateral** (or extensions of these structures in more dimensions). This is determined by the mesh.
2. The **dimension** of the cell (e.g. interval, triangle, tetrahedron). This depends on the mesh. This affects the shape of temporaries by affecting the number of coefficients in a polynomial of degree p on dimension d.
3. The **degree** of the polynomials we wish to represent on the cell. Degree k means we are dealing with P a space of k+1 dimensional polynomials on K.

We discuss the effect of the mesh and the degree together since these both affect the number of degrees of freedom per cell and their effects are linked. For example, a triangular cell has fewer nodes than a quadratic cell of the same degree and dimension.

Simplices For simplices (lines, triangles, tetrahedra), see 4.3 for examples, we know the number of degrees of freedom is the number of coefficients of a polynomial of degree p in d dimensions, which is $\binom{p+d}{d}$. I.e. for simplices:

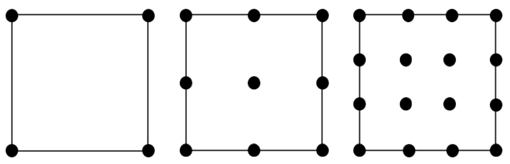
$$\text{degrees of freedom} = \binom{\text{degree} + \text{dimension}}{\text{dimension}} \tag{4.6}$$

For example for `Transpose` for the discontinuous Helmholtz 4.1, the degree p is 1 and the dimension d is 2 (from the construction of the mesh). A 1 degree polynomial in 2D has the form $a + bx + cy$ with $\binom{1+2}{2}$ which is 3 coefficients a, b and c. Hence each value in the shape tuple of the temporaries in this problem is 3.

Quadrilaterals For a quadrilateral mesh, the relationship between degree, dimension and degrees of freedom is:

$$\text{degrees of freedom} = (\text{degree} + 1)^{\text{dimension}} \tag{4.7}$$

This formula is easy to see when considering the 3D example as several layered 2D squares, each square with the appropriate number of nodes for the degree. See 4.4 for a clarifying example.


(a) Lagrange elements on quadrilaterals in 2D. From left to right, the degree being 1, 2, 3.

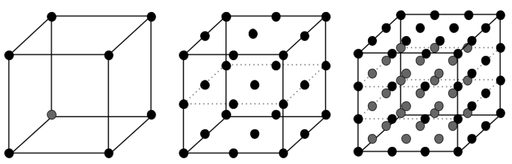

(b) Lagrange elements on quadrilaterals in 3D. From left to right, the degree being 1, 2, 3.

Figure 4.4: Lagrange elements on quadrilaterals.

We note that the popular serendipity finite element spaces are not used. The span of a serendipity space of degree p is the span of the Lagrange element of degree p together with the additional monomials $x^p y$ and xy^p.

Mixed function spaces Here we consider a mixed function space, i.e. a function space consisting of several different function spaces, all of which must be on the same mesh. The degrees of freedom for a mixed function space is the sum of the degrees of freedom of all the function spaces that are a part of it.

For example, if we have a mixed function space `W` = `FS1 * FS2`, with `FS1` and `FS2` two function spaces, then we find the degrees of freedom per cell for each of `FS1` and `FS2` using the model described above and sum these to get the degrees of freedom for `W`.

Trace spaces Trace spaces are used in hybridization so we briefly discuss the most relevant difference in terms of the degrees of freedom of these spaces from the usual spaces we have talked about above. In table 4.3 we compare the shapes for temporaries for different degrees for Discontinuous Lagrange (DG) and the trace space HDiv Trace (HDivT). All values are for a UnitSquareMesh of size 4x4.

Family	Degree	Shape example
DG	0	(1, 1)
DG	1	(3, 3)
DG	2	(6, 6)
DG	10	(66, 66)
HDivT	0	(3, 3)
HDivT	1	(6, 6)
HDivT	2	(9, 9)
HDivT	11	(36, 36)

Table 4.3: Comparing shape of temporaries for different degrees for Discontinuous Lagrange (DG) and the trace space HDiv Trace (HDivT)

We find that for the trace space, for simplices on a 2 dimensional mesh:

$$\text{degrees of freedom} = 3 * (degree + 1) \tag{4.8}$$

When test and trial functions come from different function spaces Consider a matrix `A = Tensor(a)` where `a` is a form that uses test functions from function space `FS1` and trial functions from function space `FS2`. Since this tests every trail function against every test function, the dimension of `A` will be (degrees of freedom of test space, degrees of freedom of trial space), so (degrees of freedom for `FS1`, degrees of freedom for `FS2`).

4.2.5 Evaluating on examples: Aggressive unary op nesting

This example was used to test the model on highly nested and stacked expressions. The findings from this example were used to improve the model's predictions for such operations.

This is a valuable example as effect of the stacking and nesting of expressions is the hardest one to predict, since it depends on how the compiler is written and what optimisations have already been implemented, rather than on the underlying mathematical properties.

In 4.14 we include the UFL setup code for this example. We should be able to predict to a reasonable accuracy using our model the memory needed for the temporaries just from this UFL setup code.

```
V = FunctionSpace(UnitSquareMesh(1, 1), "DG", 3)
f = Function(V)
g = Function(V)
f.assign(1.0)
g.assign(0.5)
F = AssembledVector(f)
G = AssembledVector(g)
u = TrialFunction(V)
v = TestFunction(V)

A = Tensor(u*v*dx)
B = Tensor(2.0*u*v*dx)
```

```

foo = (B.T*A.inv).T*G + (-A.inv.T*B.T).inv*F + B.inv*(A.T).T*F
result = assemble(foo)
```

Listing 4.14: Aggressive unary op nesting UFL setup code

Size and values of the shape of temporaries Let's first work out the shape and extents of the temporaries. This will be the same before and after the optimisation. The mesh is a unit square 2D mesh with triangular elements. We have the DG family with degree 3. Hence the degrees of freedom will be $\binom{degree+dimension}{dimension} = \binom{3+2}{2}$ which is 10. Hence the shape for AssembledVector temporaries will be (10) and for Tensors will be (10,10).

Before To find the number of temporaries before the optimisation, we simply add the number of temporaries for each stacked operation, which we can get from the table of number of temporaries above 4.1.

In table 4.4 we do a step-by-step outline how we make our prediction for the number of temporaries. The actual output code, labelled with the operation each line corresponds to, is included in the appendix under D.

Note that in all our derivation tables, including 4.4, the temporary names are the actual names for the relevant temporaries in the output code, not predictions of the names of the temporaries. These have been included for ease of reference to the output code.

Temp num	Temp names	Operation
2 (.,.)	T0 + t0	Tensor() A
1 (.,.)	t1	A.inv
2 (.,.)	T1 + t3	Tensor() B
1 (.,.)	t4	A.inv.T
1 (.,.)	t5	-A.inv.T
1 (.,.)	t6	B.T
2 (.,.)	t7 + t12	-A.inv.T * B.T (mat-mat multiply)
2 (.,.)	t8 + t13	B.T * A.inv (mat-mat multiply)
1 (.)	t9	AssembledVector() G
1 (.)	t10	AssembledVector() F
1 (.,.)	t11	A.T
1 (.,.)	t14	B.inv
1 (.,.)	t16	(B.T * A.inv).T
1 (.,.)	t17	(-A.inv.T*B.T).inv
1 (.,.)	t19	(A.T).T
2 (.)	t20 + t23	(-A.inv.T*B.T).inv * F (mat-vec multiply)
2 (.)	t21 + t24	(B.T * A.inv).T*G (mat-vec multiply)
2 (.,.)	t22 + t25	B.inv * (A.T).T (mat-mat multiply)
2 (.)	t26 + t27	(B.inv * (A.T).T)*F (mat-vec multiply)
1 (.)	t28	((B.T*A.inv).T*G) + ((-A.inv.T*B.T).inv*F) (add)
1 (.)	t29	((B.T*A.inv).T*G + (-A.inv.T*B.T).inv*F) + (B.inv*(A.T).T*F) (add)

Table 4.4: Step-by-step derivation of the number and shapes on temporaries for the aggressive unary op nesting, before optimisation

In total we predict 19 (.,.) + 10 (.), temporaries, giving a predicted memory requirement of temporaries as: 8 * ((19 * 10 * 10) + (10 * 10)) = 16,000 bytes. This is exactly what we get from the actual output code: our model is completely correct.

After We set out our step by step derivation of the predicted temporaries in table 4.5. Since the predictions for post-optimisation are more complex and context-specific, for each operation we also include the reasoning for the temporaries predicted. For a visual breakdown of what parts of the output code relate to which temporaries, see the actual code annotated in the appendix under D.

Temp num	Temp names	Operation	Reason
1 (.,.)	T0	Tensor() A	
1 (.,.)	T1	Tensor() B	
1 (.,.)	t0	A.inv	inverse
1 ()	t2	-A.inv	negation before mat-mat mult
1 (.,.)	t3	-A.inv.T*B.T	mat-mat mult before inv
1 (.,.)	t4	B.T*A.inv	mat-mat mult includes a tranpose
1 (.,.)	t5	(-A.inv.T*B.T).inv	inverse
1 (.,.)	t7	B.inv	inverse
1(.)	t9	(B.T*A.inv).T*G	mat-vec mult with 1 tranpose on the matrix
1 (.)	t10	B.inv*(A.T).T	mat-mat mult
1 ()	t11	(-A.inv.T*B.T).inv*F	mat-vec mult
1 ()	t12	(B.inv*(A.T).T)*F	mat-vec mult
0		add all parts	final operation (straight to output buffer)

Table 4.5: Step-by-step derivation of the number and shapes on temporaries for the aggressive unary op nesting, after optimisation

Total number of temporaries predicted: 7 (.,.) + 2 (.) + 3 (). Thus, for the total memory size of temporaries we estimate: 8 * ((7*10*10) + (2*10) + (3*1)) = 5784 bytes. This is exactly what we find from the actual output: again our model is completely correct.

For this example we have found a reduction from 16,000 to 5784 bytes required for the temporaries, i.e. a reduction of almost 64%.

4.2.6 Evaluating on examples: Hybridization

Hybridization is an important use case for Slate, as discussed in 2. It is a representative example of the types of workloads handled by Slate. As such, it is valuable to consider the effects of our optimisation on a hybridization example to show the effects of the optimisation in real-world applications.

Below we complete a step by step analysis of one step within a hybridization example adapted from [56]. This allows us to show our model in the more representative scenarios without laboring the point with very similar, long derivations of all the series of operations taken in hybridization.

In 4.15 we include the UFL setup code for our example.

```
degree = 1
hdiv_family = "RT"
mesh = UnitSquareMesh(6, 6, quadrilateral=False)
RT = FunctionSpace(mesh, hdiv_family, degree)
DG = FunctionSpace(mesh, "DG", degree - 1)
W = RT * DG
sigma, u = TrialFunctions(W)
tau, v = TestFunctions(W)
n = FacetNormal(mesh)

# Define the source function
f = Function(DG)
x, y = SpatialCoordinate(mesh)
f.interpolate((1+8*pi*pi)*sin(x*pi*2)*sin(y*pi*2))

# Define the variational forms
a = (dot(sigma, tau) - div(tau) * u + u * v + v * div(sigma)) * dx
L = f * v * dx - 42 * dot(tau, n)*ds

# Hybridized solution
w = Function(W)
params = {'mat_type': 'matfree',
          'ksp_type': 'preonly',
          'pc_type': 'python',
          'pc_python_type': 'firedrake.HybridizationPC',
          'hybridization': {'ksp_type': 'preonly',
                            'pc_type': 'lu'}}
solve(a == L, w, solver_parameters=params)
```

Listing 4.15: Extract from simple hybridization example UFL setup code [56]

As mentioned above, we will analyse one of the series of operations that form part of the hybridization example. The series of operations we will consider are those that assemble the Schur complement operator and right-hand side.

In listing 4.16 we include modified extracts from the the hybridization method to show what kinds of tensors and operations we are dealing with, such that the temporaries created can be predicted and understood.

```
# Determine shape of K
tdegree = 0  # Comes from RT1
TraceSpace = FunctionSpace(mesh, "HDiv Trace", tdegree)
gammar = TestFunction(TraceSpace)
n = ufl.FacetNormal(mesh)
sigma = TrialFunctions(V_d)[self.vidx]
Kform = (gammar('+') * ufl.jump(sigma, n=n) * ufl.dS)
K = Tensor(Kform)

# Determine shape of Atilde
broken_elements = ufl.MixedElement(...)
V_d = FunctionSpace(mesh, broken_elements)
arg_map = {test: TestFunction(V_d), trial: TrialFunction(V_d)}
Atilde = Tensor(replace(self.ctx.a, arg_map))

# Determine shape of AssembledVector b_r
self.broken_residual = Function(V_d)
b_r = AssembledVector(self.broken_residual)

# Schur right-hand side
K * Atilde.inv * b_r

# Schur complement operator
K * Atilde.inv * K.T
```

Listing 4.16: Modified extracts from the hybridization example. [56] Comments have been added for clarity.

Size and values of the shape of temporaries From 4.15 we see that the mesh is 2D and consists of triangular elements. `W` is a mixed function space and our trial, test and solution function, `w`, all live in this space. We calculate the degrees of freedom:

- For `RT1`: $\binom{1+2}{2} = 3$,
- For `DG` $= \binom{0+2}{2} = 1$,
- Hence we find the degrees of freedom for `W` $=$ `RT`*`DG` $= 3 + 1 = 4$.

From 4.16, we see that `V_d` is a space like `W` but discontinuous, it is the space of 'broken' `RT1` x 'broken' `DG0`. It has the same degrees of freedom as `W`, so 4.

`K` is the Tensor of the `Kform` which is a form that uses test functions from the trace space `RT1` and trial functions from the `V_d` space. Since this tests every trail function against every test function, the dimension of `K` will be (degrees of freedom of test space, degrees of freedom of trial space), so (3,4).

`Atilde` is the bilinear form defining the matrix. This is in space `W`, which we have above determined to have 4 degrees of freedom. Hence temporaries storing `Atilde` will have the shape (4,4). Also, this means `Atilde.inv`, for example, will also be of shape (4,4).

`b_r` is an AssembledVector living in the `V_d` space, hence will have shape (4).

We can derive that the multiplication of `K * Atilde.inv`, i.e. a (3,4) matrix with a (4,4) matrix will result in a (3,4) matrix. Similarly, a (3,4) matrix (`K*Atilde.inv`) multiplied by a vector of shape (4) (`b_r`) will yield a vector of shape (3).

Schur right-hand side, `K * Atilde.inv * b_r`

Before the optimisation, from 4.6 we see that in total we predict 7 (.,.) + 3 (.), for a total memory of 8*(3*4*4 + 4*3*4 + 4 + 2*3) = 848 bytes. This is exactly what we find is the true value from the actual outputted code.

Temp num	Temp Name	Temp shape	Operation
2 (.,.)	T1 + t3	(3,4)	Tensor() K
2 (.,.)	T0 + t0	(4,4)	Tensor() Atilde
1 (.)	t6	(4)	AssembledVector() b_r
1 (.,.)	t1	(4,4)	Atilde.inv
2 (.,.)	t4 + t5	(3,4)	K * Atilde.inv
2 (.)	t7 + t8	(3)	(K * Atilde.inv) * b_r

Table 4.6: Derivation of the number and shapes on temporaries for the schur right-hand side `K * Atilde.inv * b_r`, before optimisation

After the optimisation, from 4.7 we predict 3 (.,.) + 1 (.) for a total memory of 8 * (2*4*4 + 3*4 + 4) = 384 bytes, which is what we find.

Temp num	Temp names	Temp shape	Operation
1 (.,.)	T1	(3,4)	Tensor() K
1 (.,.)	T0	(4,4)	Tensor() Atilde
1 (.,.)	t0	(4,4)	Atilde.inv
1 (.)	t2	(4)	K * Atilde.inv
0			(K * Atilde.inv) * b_r (straight to output)

Table 4.7: Derivation of the number and shapes on temporaries for the schur right-hand side `K * Atilde.inv * b_r`, after optimisation

So the difference in memory for the Schur right-hand side has been a reduction from 848 to 384 bytes, i.e. a reduction by almost 55%.

Schur-complement, `K * Atilde.inv * K.T`

Before the optimisation, from 4.8 we predict 10 (.,.) for a total memory size of 8*(4*4*4 + 6*3*4) = 1088 bytes, which is what we find.

Temp num	Temp Name	Temp shape	Operation
2 (.,.)	T0 + t1	(3,4)	Tensor() K
2 (.,.)	T1 + t0	(4,4)	Tensor() Atilde
1 (.,.)	t2	(4,4)	Atilde.inv
2 (.,.)	t4 + t5	(3,4)	K * Atilde.inv
1 (.,.)	t6	(4,3)	K.T
2 (.,.)	t7 + t8	(3,3)	(K * Atilde.inv) * K.T

Table 4.8: Derivation of the number and shapes on temporaries for the schur complement, `K * Atilde.inv * K.T`, before optimisation

After the optimisation, from 4.9 we predict 3 (.,.) + 1(.), for a total memory of 8 * (2*4*4 + 3*4 + 4) = 384 bytes, which is what we actually get from the output too.

Temp num	Temp Name	Temp shape	Operation
1 (.,.)	T0	(3,4)	Tensor() K
1 (.,.)	T1	(4,4)	Tensor() Atilde
1 (.,.)	t0	(4,4)	Atilde.inv
1 (.)	t2	(4)	K * Atilde.inv
0			(K * Atilde.inv) * K.T (straight to output)

Table 4.9: Derivation of the number and shapes on temporaries for the schur complement, `K * Atilde.inv * K.T`, after optimisation

The difference in memory needed for temporaries for the Schur-complement has been a reduction from 1088 to 384 bytes, i.e. a reduction by almost 65%.

Summary of the entire hybridization example
We have seen large reductions in memory for the above parts of hybridization. Below we summarise the overall effects of the optimisation by considering the differences in temporaries before and after the optimisation for the entire hybridization and solving process.

Before:
Total memory taken up by temporaries: 848+1008+1080+936+776= 4648.
Total number of temporaries: 93.
Shapes: {61 (.,.), 32 (.)}.

After:
Total memory taken up by temporaries: 384+384+408+416+280= 1872.
Total number of temporaries: 36.
Shapes: {22 (.,.), 9 (.), 5 ()}.

We see far fewer temporaries being created of all sizes, and a memory reduction from 4648 to 1872 bytes - almost 60% reduction in the memory required to store temporaries.

Such memory saving can quickly add to up be very significant, especially for larger, more complex examples.

4.3 Evaluation of optimisation to remove unneccessary temporaries

In this part of the project we aimed to reduce the number of temporaries needed for performing Slate operations, thus reducing the working set size for the resulting instructions.

This was achieved through manipulations of the GEM expression DAG, namely through the elimination of ComponentTensors. Performing this transformation here has been inspired by the use of such a transformation in TSFC for easier code generation.

On balance, we consider this optimisation to have been successful since it led to many unneccessary temporaries not being created when performing operations on tensors.

Below we discuss some of the key limitations and strengths of our optimisation.

4.3.1 Limitations

Recomputation of differently substituted expressions The most notable disadvantages of our optimisation is it can result in the recomputation of expressions that are substituted differently. An example of this is given in 4.2.1. In this example we discuss how, for the example of calculating `-B*-A*-A*-B`, `-B` is calculated twice and is stored in a different temporary each time, thus performing unneccessary computations and creating unneccessary temporaries. Because of this, one may wish to turn off this optimisation for certain types of operations, which can be done by passing a flag in the function that preprocesses the GEM DAG.

Memory not reduced to optimal amount The elimination of `ComponentTensors` has not led to preventing memory from being used unneccessarily for storing temporaries. In 4.2.1, we discuss how an odd number of tranposes results in a vector temporary rather than a scalar one. This unneccessarily uses memory due to a lack of nesting of `for` loops in the code generated. This *is* done in the case where there are an even number of transposes. Hence, we see this optimisation has not reduced the memory needed to the optimal levels, and there are still improvements that can be done. As the difference in memory needed is not large, its likely not a significant difference.

4.3.2 Strengths

Reduction of memory needed This optimisation achieved the objective of reducing the amount of memory needed to store temporaries. It achieved a reduction in memory of 55% - 65%, with the representative hybridization example discussed in 4.2.6 having a reduction by 60%.

Most operations use the minimal number of temporaries possible In many instances, the number of temporaries required for the given operation is now the minimum number possible, taking into account the one temporary of form Tn that is used to store the tensor itself. One such example is transposition, which formerly needed 3 temporaries and now only needs the 1 neccessary Tn temporar.

Simpler expression graphs Through the removal of ComponentTensors in the GEM expression DAGs, we have greatly simplified these DAGs, thereby making them easier to understand and work with.

4.4 Evaluation of model for amount of memory needed for temporaries

4.4.1 Limitations

Possible unaccounted-for stacking effects We found that the number and types of temporaries could be determined solely by the operations performed on a tensor, however there were stacking effects which resulted in different numbers (and sometimes shapes) of temporaries needed if operations were performed in a certain order. We investigated these stacking effects in a variety of examples. We note that there may be combinations of operations that are not included in these examples which stack in a way our model does not predict. This would result in slightly different memory requirements for the tensors.

Predictions need to be done manually A disadvantage of our model is the time needed to manually apply the model to the given case, especially the derivation of the number and shapes of temporaries from the operations performed. This process is tedious and long, so if this model is used frequently or for large, complex examples, then automating this model using a computer program might be beneficial.

Does not cover all cases For things like mesh and finite element, our model includes only common choices and those related to this project, as this was sufficient for our use case of the model. There are other values of these that the model has not been built or tested with. These might be of interest to some users, and their inclusion is a possibility for future work.

4.4.2 Strengths

Predictions close to the real values We have found that the model's predictions tend not to be too far off the correct answer – the accuracy is sufficient for the purpose of making predictions for roughly the memory required for a set of operations. Therefore, the model is also suitable for comparing two different sets of operations, such as those for two different approaches to a problem, or before and after an optimisation or other code change.

Insight into effect of choice of mesh and finite element The model gives insight into why the number, shapes and extents of temporaries are needed, and hence how the choices of mesh and finite element can affect the amount of memory needed for temporaries.

4.5 Summary

In this chapter we investigated the specific effects that a reduction in temporaries has in reducing the memory overhead. This was done to better understand the effects of our optimisation on the memory required. We did this by developing a model for the amount of memory required by the temporaries to perform operations on tensors.

We hypothesised that the number of temporaries could be determined solely through the set of operations that were being performed. We found that by assuming this, we were able to get accurate predictions for the number of temporaries for the examples we tested. When analysing what affects the temporaries being created, we found no other factors affecting the number of temporaries. Hence, there is insignificant evidence to say that we can't determine the number of temporaries soley through the operations.

We also hypothesised that our model can accurately predict the number of temporaries required for a sequence of operations from the temporaries needed for each basic operation that needs to be performed as part of that set of operations. If this hypothesis were true it would also imply that our hypothesis in the previous chapter 3.3, is also true: that by optimising the basic building-block operations we would be optimising the evaluation of more complex expressions too. We experimented with predicting the number of temporaries for a sequence by adding the contributions from each basic operation performed. We found that this yielded large overpredictions, both before and after our optimisation. This led us to discover stacking effects, in which some temporaries are re-used by the compiler from previous optimisations. We explored these effects and found that they impact not only the number of temporaries, but also the shape of the temporaries. In summary, we conclude there is no evidence to suggest our hypothesis is wrong, hence our model predicts the number of temporaries from the basic operations only.

We found that the shape of the temporaries depends on the type of tensor being stored (e.g. a 2-tuple for shape for matrices, 1-tuple for vectors). However, the type of tensor being stored, could be affected by the stacking effects, resulting in a differently shaped temporary to what was expected.

We found that since finite element families can differ by the polynomial space P, they can effect the extent of the temporary in each dimension. The mesh and degree had interconnected effects, which we discuss in more detail in 4.2.4.

From the work done in this chapter and by applying the model for various examples before and after our optimisation we were able to better understand the effects and limitations of our optimisation. For example, we found that our optimisation did not remove all unneccessary memory associated with temporaries due to it resulting in the need to recompute differently substituted expressions.

We conclude this part of the project by saying that, on balance, we consider that our optimisation has been successful – it led to many unneccessary temporaries to be removed, yielding a reduction in memory of 55% - 65% as per our objectives.

Chapter 5

Part 2: Structured Sparse Local Tensors

5.1 Introduction

There are many zeros in the tensors involved in the assembly process of the finite element method - both in the local and the global tensors. It is very inefficient to store these zeros and to perform calculations on them.

In Firedrake, the sparsity of the global matrices is already accounted for and handled through the use of sparsity-aware storage formats such as compressed row storage. However, the local tensors are also sparse, and this is not accounted for in Firedrake.

The aim of this part of the project was to implement sparse local tensors. This task was broken down into two separate stages:

1. **Implement sparsity in local kernels**
   This involved ensuring that:
   - The space taken up by the local tensor was only the space of the non-zeros. We will call the structure storing these values the *buffer*. This also meant needing to find a mapping from the buffer back to the original *tensor*, i.e. a mapping of where the non-zeros go in the original tensor.
   - The operations on the tensor where performed only on the zeros. This was already being done.
2. **Propagate the local tensor sparsity**
   Find a way to propagate the sparsity information to things using the local tensor such that the sparsity can be taken advantage of.

To aid in the understanding of the work done in this chapter, we now briefly discuss the types of sparsity we encounter, why local tensor sparsity arises and the different ways one can index a sparse tensor.

5.2 Context: Sparsity and indexing strategies

5.2.1 Structured and unstructured sparsity

It is important to note that local tensors have a different type of sparsity to that of the global tensors. The sparsity of global tensors is unstructured, in that its known that it is mainly composed of zeros but one can't derive where the non-zeros are. This is because the sparsity is driven by the connectivity of the mesh. Hence, to work with such tensors, we need to store the non-zero values as well as a mapping to say where each non-zero value is within the tensor.

Local tensors, on the other hand, have structured sparsity, meaning that the position of the non-zeros can be derived through a closed-form formula.

5.2.2 How local tensor sparsity arises

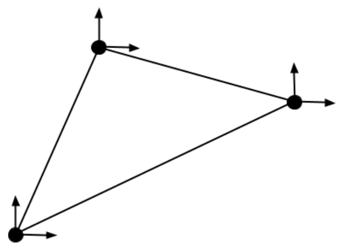


Figure 5.1: Element of the $(\mathrm{CG1})^2$ vector space.

We will demonstrate the source of the sparsity through an example.

Let $\bar{e}$ be the canonical basis of $\mathbb{R}^2$, i.e. $\begin{pmatrix}1 & 0\end{pmatrix}^T$, $\begin{pmatrix}0 & 1\end{pmatrix}^T$.

Assume a $(\mathrm{CG1})^2$ vector function space, call it V.

Hence our elements are triangles, we evaluate two vector components at each node and there is a node at each of the three vertices of the triangle for a total of 6 degrees of freedom per cell (and hence 6 basis functions), see figure 5.1.
Take the vector basis function for V as

$$\bar{\phi}_{i\alpha} = \phi_i \bar{e}_\alpha \tag{5.1}$$

with ϕ_i a scalar value, $\bar{e}_\alpha$ the αth canonical basis of $\mathbb{R}^2$, $i = \{0, 1, 2\}$ the index for the vertices and $\alpha = \{0, 1\}$ the index for the x and y components at each node.
If we take the βth entry of the $i\alpha$th basis vector:

$$\bar{\phi}_{i\alpha}|_\beta = \phi_i \bar{e}_\alpha|_\beta = \phi_i \delta_{\alpha\beta} \tag{5.2}$$

For example, letting $\phi_0 = x$, $\bar{\phi}_{00} = \phi_0 \bar{e}_0 = \begin{pmatrix}x & 0\end{pmatrix}^T$ and $\bar{\phi}_{00}|_0 = x$, $\bar{\phi}_{00}|_1 = 0$.
Take a local tensor, $\mathtt{A}$, defined as

$$\mathtt{A}_{ij\alpha_1\alpha_2} = \int \bar{\phi}_{i\alpha_1} \cdot \bar{\phi}_{j\alpha_2} dx \quad \forall i, \alpha_1, j, \alpha_2 \tag{5.3}$$

For given i, j, α_1, α_2,

$$\mathtt{A}_{ij\alpha_1\alpha_2} = \sum_\beta \phi_i \delta_{\alpha_1\beta} \phi_j \delta_{\alpha_2\beta} \tag{5.4}$$

which is only non-zero when $\alpha_1 = \alpha_2 = \beta$. This is known as *delta cancellation*.

This results in the local tensor having a form like that in figure 5.2, or the transpose of that, which would look like the checkerboard seen in 5.3a.

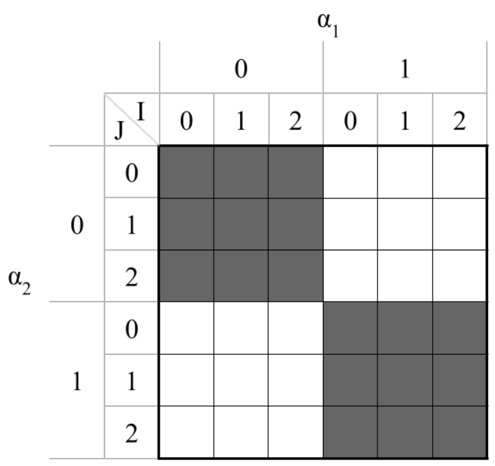
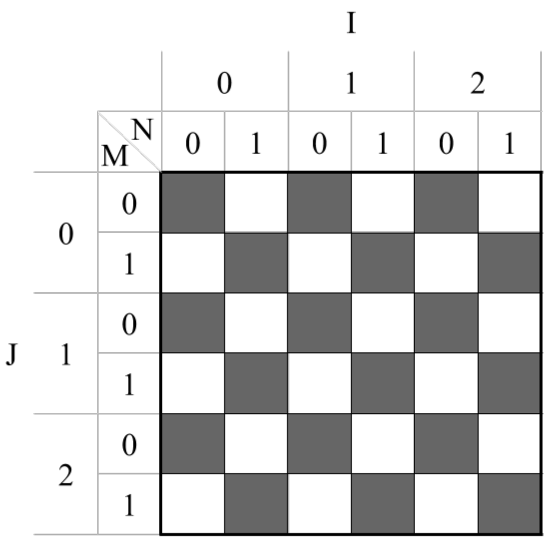


Figure 5.2: Local matrix zeros structure. The dark grey represents non-zero entries, while the white represents zero entries.

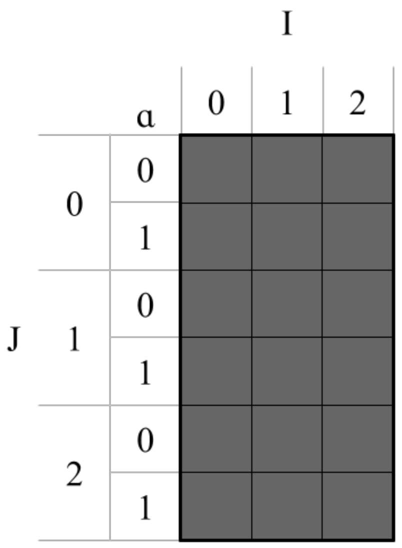
5.2.3 Different indexing strategies

We will demonstrate the different ways of indexing a structurally sparse tensor through a simple example. For brevity we will refer to matrices indexed by n indices as *n-tensors*. Take `A`, a 6x6 a 4-tensor indexed by `I`, `J`, `M` and `N`, as shown in figure 5.3a. Note how the non-zeros occur only when `M` and `N` take on the same value. Since we are only interested in the non-zero values, we can represent the `M` and `N` value by a single index, call it α. So now we see how we can represent `A` as a 3-tensor, indexed by `i`, `j` and α, as in figure 5.3b.

From 5.3b we can easily see how we can think of `A` as a 2-tensor, accessible by row and column index, call the indices `R` and `C`. We observe how in this case `R`$= 2j + \alpha$ and `C`$= 2i + \alpha$.



(a) 4-tensor view of `A`, indexed by `I`, `J`, `M` and `N`.



(b) 3-tensor view of `A` indexed by `i`, `j` and α.

Figure 5.3: Different ways of indexing a 6x6 structurally sparse local tensor `A`. The dark grey represents non-zero entries, while the white represents zero entries.

5.3 Implementing sparsity in local kernels

To motivate and test our implementation we consider the example in 5.1. Note how `a`, and thus `A`, is the same as 5.3, the `A` in the sparsity discussion above. Hence we see that this tensor is going to be structurally sparse.

In 5.1, `mat_type='aij'` indicates that we are assembling matrix A as a monolithic matrix, as opposed to the default for this case which is as a block matrix.

```
mesh = UnitTriangleMesh()  # A mesh of a single triangle
V = VectorFunctionSpace(mesh, "CG", 1)
u = TrialFunction(V)
v = TestFunction(V)
a = inner(u,v) * dx
A = assemble(a, mat_type='aij')
```

Listing 5.1: UFL code for structured sparsity example

5.3.1 Before

In 5.2 below, we include relevant extracts of the original loopy kernel for our example. For the full original loopy kernel, see E.1.

```
ARGUMENTS:
A: type: np:dtype('float64'), shape: (6, 6), dim_tags: (N1:stride:6, N0:stride:1) aspace:
    global
```

```
coords: type: np:dtype('float64'), shape: (6), dim_tags: (N0:stride:1) aspace: global
---------------------------------------------------------------------------
DOMAINS:
{ [j0] : 0 <= j0 <= 2 }
{ [k0] : 0 <= k0 <= 2 }
{ [ip] : 0 <= ip <= 2 }
{ [j0_0] : 0 <= j0_0 <= 2 }
{ [k0_0] : 0 <= k0_0 <= 2 }
{ [j0_1] : 0 <= j0_1 <= 2 }
{ [k0_1] : 0 <= k0_1 <= 2 }
---------------------------------------------------------------------------
INSTRUCTIONS:
...
for k0_1, j0_1
    A[0 + j0_1*2, 0 + k0_1*2] = A[j0_1*2, k0_1*2] + t7[j0_1, k0_1]  {id=insn_6, priority
    =2}
    A[1 + j0_1*2, 1 + k0_1*2] = A[1 + j0_1*2, 1 + k0_1*2] + t7[j0_1, k0_1]  {id=insn_7,
    priority=1}
end k0_1, j0_1
```

Listing 5.2: Extracts of original local kernel. '...' indicates that there is more content that has not been included here for brevity.

ARGUMENTS are data that carry information into and out of the loopy kernel. For our example we have an argument named A which corresponds to the local tensor whose sparsity we want to take advantage of. The shape of A is 6x6, and it has the same shape and sparsity structure as the tensor discussed in section 5.2.3. The argument coords represents the 6 coefficients of the nodes, and is just a normal dense tensor.

The DOMAINS section states the values each index can take.

We see that the index accesses into A are of the form $A[2j0\_1 + \alpha, 2k0\_1 + \alpha]$, where α, is unrolled into separate statements rather than being included as a loop over an index. This matches the 2-tensor indexing we saw in 5.2.3. Considering the 3-tensor view in 5.3b, which has indices i, j and α, we can deduce the correspondence between the code indices and those used in our indexing example as: j0_1 maps to j, and k0_1 maps to i. Since α is implicitly included, there is no named index to which it corresponds.

To summarise, we are storing a 6x6 tensor in memory, which in this case has half its values as zeros. This is inefficient. In terms of work done on the tensor, at the local kernel stage: from the 2-tensor indexing used we are only accessing non-zero values, hence we are not performing unneccessary calculations on zeros at this stage - thus we are taking advantage of the sparsity.

When we use that local tensor somewhere else, such as in global assembly, the information about sparsity is lost. Hence, we waste memory storing many zeros, and need to do calculations on all values, including on zeros.

5.3.2 After

In 5.3 we have the key extracts from the final local loopy kernel, after our changes have been implemented. For the full version of the final local loopy kernel, see E.2.

```
ARGUMENTS:
A: type: np:dtype('float64'), shape: (2, 3, 3), dim_tags: (N2:stride:9, N1:stride:3, N0:
    stride:1) aspace: global
coords: type: np:dtype('float64'), shape: (6), dim_tags: (N0:stride:1) aspace: global
---------------------------------------------------------------------------
DOMAINS:
...
{ [j1] : 0 <= j1 <= 1 }
---------------------------------------------------------------------------
INSTRUCTIONS:
...
for j1, j0_1, k0_1
      A[0 + j1, 0 + j0_1, 0 + k0_1] = A[j1, j0_1, k0_1] + t7[j0_1, k0_1]  {id=insn_6,
    priority=1}
```

```
end j1, j0_1, k0_1
```

Listing 5.3: Extracts of final local kernel. '...' indicates that there is more content that has not been included here for brevity.

Argument shape We note that the number of entries in `A` is 18, exactly the number of non-zeros needed. The buffer we desire should have total size equal to the number of non-zeros, but we there is no clear preferrable shape to achieve this (e.g. for our case we could have had shape (18) or (9,2)).

Since the total size needed can be calculated from the domains of the indices indexing into the local tensor, we chose to have the shape length equal to the number of indices, with the extent in each dimension equal to the number of possible values each index can take (knowing that the values indices can take are integers). By doing this, it not only simplifies the process of indexing into the local tensor (since now we can just index into it using the indexes directly with no further transforms needed) but it also means we retain some information about the number and extents of the indexes.

We have made it such that the indices affected by delta elimination are the slowest changing, and thus for our choice of shape format will be the ones in the first positions in the shape tuple. This has been done to bring the local tensor into the order expected by PyOP2 for global assembly.

5.3.3 Implementation details

The implementation involved creating a new type of GEM node, which we called a `Structured-SparseVariable`.

Initial approach Our initial approach saw us attempting to have the kernel builder initialise the object representing `A` as a `StructuredSparseVariable`. This would behave like the `Variable` node and its `FlexiblyIndexed` wrapper that are currently being used, such that optimisations could be applied in the usual way. However as delta eliminations and other possible index changing operations were performed on it, it would keep a record such that it would have all the information during the code generation stage to map between the buffer and the original tensor.

We ultimately chose against this as this would cause unneccessary duplication and complication of the code involving in determining all the parts of the kernel, especially the optimisations.

Using FlexiblyIndexed nodes We found that we could repurpose an existing construct. A `FlexiblyIndexed` node has an attribute `dim2idxs` which contains all the information about indexing the tensor: the offset, index and stride. We give an example of a `FlexiblyIndexed` node in 5.4.

The information in `dim2idxs` has already been affected by all the index-changing optimisations. We only care about what effect the optimisations had on the indexes and not about what the optimisations applied were themselves. Hence, we can simply use the information in `dim2idxs` to determine the mapping from buffer to original tensor.

```
FlexiblyIndexed(
    Variable('A', (6, 6)),
    ( # dim2idxs below
        (0, ((Index(3), 2), (Index(4), 1))),
        (0, ((Index(5), 2), (Index(4), 1)))
    )
)
```

Listing 5.4: `FlexiblyIndexed` node for the local tensor `A` from our structures sparsity example. Refering to Index(n) as `In`, this corresponds to the indexing `A`[0 + `I3*2` + `I4*1`, 0 + `I5*2` + `I4*1`]. The repeated index (`I4`) imposes a delta cancellation.

Solution We decided to use the `FlexiblyIndexed` nodes.

To facilitate this, we undid the unrolling of the index corresponding to α, and thus added in an explicit loop over a new index `j1`, with $0 \leq j1 \leq 1$, as seen in 5.3.

Our solution involved keeping the initial part of the code generation process the same, thus creating the required `FlexiblyIndexed` nodes: declaring an ordinary dense tensor as before, applying all the optimisations desired and creating the expression tree representing the kernel. Then, after this expression tree is fully completed, and before it is traversed to generate the actual code, we pass through the list of arguments and wrap the ones corresponding to local tensors (these will be the `FlexiblyIndexed` nodes that have had delta cancellations applied to them) in `StructuredSparseVariables` so that the later code generation stage knows to treat them as sparse.

Hence, ultimately the `StructuredSparseVariable` node works like a wrapper for a `Flexibly-Indexed` node. The `StructuredSparseVariable` captures the notion of a dense tensor with delta cancellations applied to it and serves to enter a different code path in the generation of code for sparse local tensors.

We perform all the structured-sparse specific actions at the code generation stage, i.e. we calculate and set the shape of the argument, as well as setting the indices into the buffer. The code generation stage is the earliest and most convenient time at which we have all the information we need to create the full structured-sparse object.

5.4 Propagating the local tensor sparsity

Once a local kernel is created, it must either be assembled into the global matrix or have operations performed on it through Slate. Since the common use case is simply to assemble the local tensor, this is the one we focused on in this project. An extension to this project would therefore be to extend Slate to handle such sparse tensors, see 6.2 for more.

We will now discuss how global assembly, how it is currently performed in Firedrake and what would need to be changed to support structured sparse local tensors. We also discuss PyOP2, as this is the part of Firedrake responsible for taking the local kernels, efficiently executing these on each element in the mesh and performing global assembly with them.

5.4.1 Global assembly procedure details

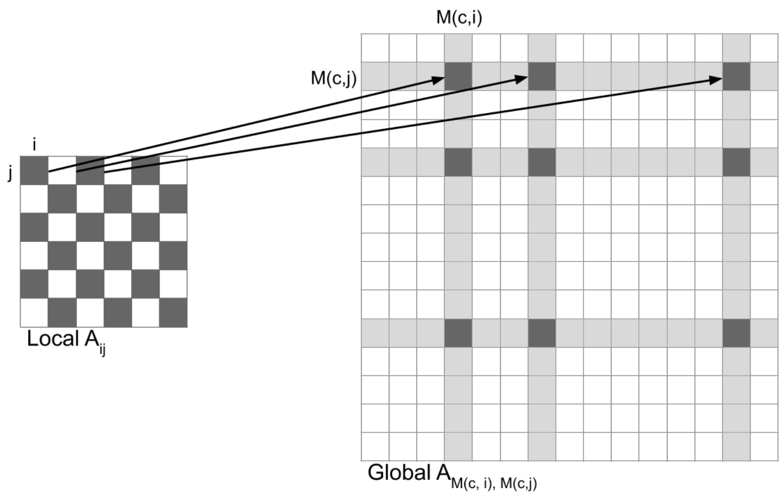


Figure 5.4: Visual representation of the global assembly procedure. The dark grey represents non-zero entries, while the white represents zero entries.

In 5.5, we include the high-level view of the global assembly procedure for our ongoing example. A visual representation of this is included in 5.4.

At each cell we calculate the cell's contribution (i.e. the local tensor) to the global tensor. The (i, j)th entry of the local tensor for cell `c` is added to $(M(c, i), M(c, j))$th entry of the global tensor.

```
A_ij = 0
for c in cells:
    for 0 <= i < N:
        for 0 <= j < N:
            $A_{M(c,i),M(c,j)} += \int_c (\Phi_i \cdot \Phi_j)|J|\,dx$
```

Listing 5.5: Global assembly procedure for $A_{ij} = \int_\Omega \phi_i \cdot \phi_j \, dx$.

where

- N is the number of nodes per element, in our case N=6.
- |J| is the absolute value of determinant of the Jacobian matrix given by $J_{\alpha\beta} = \frac{\partial x_\alpha}{\partial X_\beta}$ with X the local coordinates, x the global coordinates.
- $M(c, \zeta)$ the cell-node map giving the global node number for the local node ζ in cell c.

5.4.2 PyOP2 concepts

Here we will include a brief discussion of key PyOP2 concepts related to the global assembly process. [57, 58]

- **Set**: Some number of mesh entities or degrees of freedom.
- **Map(iterset, toset, arity, values, vertical_offset)**: Connectivity between two sets.
- **DataSet(iterset)**: A data set. Subclass of `Set`. Used in `Dat` to specify the dimension of the data.
- **Dat**: Associates a `DataSet` with data.
- **Sparsity(dsets, maps)**: Represents the non-zero structure of a matrix. Defined from `DataSets` for the left and right function spaces it is mapping between. The non-zero structure is derived from the outer product of the `Map` objects. The number of rows is equal to the size of the input `Set`, the number of columns is equal to the size of the output `Set`.
- **Mat(sparsity)**: This represents a matrix and can be viewed as something that takes in a `Dat` on a `Dataset` and outputs a `Dat` on a different `Dataset`. `Mats` are defined on a `Sparsity` pattern and have a value for each entry in the `Sparsity`. The size of the `Mat` is defined from the dimensions of input and output `Datasets`. `Mats` are stored using Compressed Sparse Row (CSR) format.

To aid in understanding, we have included below an example modified from Bercea 2017 [57]. Take the mesh as in figure 5.5.

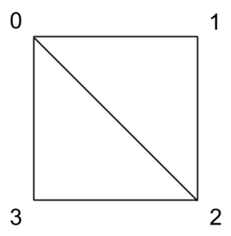


Figure 5.5: Simple mesh with 2 triangular cells and 4 vertices.

Below in listing 5.6 we include some simple example uses of the PyOP2 constructs outlined above.

```
vertices = Set(4)
verticesDataSet = DataSet(vertices, dim=2) # 2 values per vertex for the x & y coordinates
coordinates = Dat(verticesDataSet, [[0, 1], [1, 1], [1, 0], [0, 0]])

cells = Set(2)
```

```
cell2vertices = Map(cells, vertices, 3, [[0, 1, 2], [0, 2, 3]])

sparsity = Sparsity((verticesDataSet, verticesDataSet), [(cell2vertices cell2vertices)])
matrix = Mat(sparsity, dtype=float)
```

Listing 5.6: Simple example of using PyOP2 objects

PyOP2 parallel loops (`par_loop`), which are just invocations of a kernel, perform global assembly. The input to these is the kernel to be executed, the iteration set (set over which the kernel should be executed) and access descriptors, `Args`.

`Args` are made from a data object (e.g. `Dat`, `Mat`), access modes (saying how the kernel will access the data) and optionally maps which dictate how the `Arg` is to be accessed. The `Args` are the global data structures which will be accessed by the kernel. [57]

PyOP2 uses petsc4py to access sparsity formats and methods for adding in local contributions.

5.4.3 Assembly implementation in PyOP2

As mentioned above, it is the PyOP2 `par_loops` that perform global assembly. In our example, in the `par_loops`, the kernel is of the form $inner(u, v) * dx$ and is the local kernel we discussed earlier. The iteration space is over cells, since dx represents a cell integral.

A local tensor is assembled for each element in the iteration set using the kernel. The wrapper function then adds these local contributions to the global tensor. The global matrix is represented by a PyOP2 `Mat`.

The non-zero structure for the global matrix is defined using local-to-global degree of freedom mappings: the row and column maps (`rowmap` and `colmap` respectively). These store the mapping from the local to the global degrees of freedom.

The sparsity of the global matrix is built by doing the outer product of `rowmap` and `colmap`. This means that PyOP2 assumes there are values for all combinations of these maps. This is equivalent to treating the local tensor as dense. We can see this in 5.7, which is a modified extract from the original code generated by PyOP2 to perform global assembly. For the full code, see E.3.

```
void wrap_form00_cell_integral_otherwise(int32_t const start, int32_t const end, Mat const
     mat0, double const *__restrict__ dat0, int32_t const *__restrict__ map0)
{
  ...
  double t1[3 * 2 * 3 * 2];  // Dense local tensor

  for (int32_t n = start; n <= -1 + end; ++n)
  {
    // Local tensor initialised to zero
    for (int32_t i2 = 0; i2 <= 2; ++i2)
      for (int32_t i3 = 0; i3 <= 1; ++i3)
        for (int32_t i4 = 0; i4 <= 2; ++i4)
          for (int32_t i5 = 0; i5 <= 1; ++i5)
            t1[12 * i2 + 6 * i3 + 2 * i4 + i5] = 0.0;

    // Packing coords
    for (int32_t i0 = 0; i0 <= 2; ++i0)
      for (int32_t i1 = 0; i1 <= 1; ++i1)
        t0[2 * i0 + i1] = dat0[2 * map0[3 * n + i0] + i1];

    // Inlined local kernel
    ...
    for (int32_t form_k0_1 = 0; form_k0_1 <= 2; ++form_k0_1)
      for (int32_t form_j0_1 = 0; form_j0_1 <= 2; ++form_j0_1)
      {
        t1[12 * form_j0_1 + 2 * form_k0_1] = t1[12 * form_j0_1 + 2 * form_k0_1] + form_t7
    [3 * form_j0_1 + form_k0_1];
        t1[...] = t1[...] + form_t7[3 * form_j0_1 + form_k0_1];
      }

    // Add local contributions from t1 to global matrix mat0
```

```c
    MatSetValuesBlockedLocal(mat0, 3, &(map0[3 * n]), 3, &(map0[3 * n]), &(t1[0]),
     ADD_VALUES);
  }
}
```

Listing 5.7: Modified extracts from original code generated from PyOP2 wrapper. Comments beginning with // have been inserted for clarity. '...' indicates that there is more content that has not been included here for brevity.

The call to `MatSetValuesBlockedLocal` is what adds the local contribution to the global matrix. The first argument `mat0` is the global matrix. The 2nd argument '3' indicates that 3 rows will be affected in the global matrix, and the indexes of these in the global matrix are given by argument 3. In this case the rows affected will be map0[3*n], map0[3*n]+1 and map0[3*n]+2. Arguments four and five are similar but these indicate the columns that will be affected. Hence we are adding a 3x3 array into `mat0`. The sixth argument indicates the first address where the values to add will be taken from. In this case, this just means the first entry of t1, our local tensor. Implicitly in mat0 there is information that says that a 2x2 block of values should be taken from t1.

5.4.4 Adapting PyOP2 assembly to support structured sparse local tensors

How this can be done

As mentioned above, PyOP2 currently assumes it knows what the local data structure is and that it is dense. We would need to change it to accept as input a local data structure and a way to work with that data structure. This way we could relay the information of sparsity such that we remove unneccessary calculations on zeros.

With this information we would adapt the way we build the sparsity that the global matrix is built upon, such that only non-zeros are assembled.

The information we would need to pass to PyOP2 would be that within the `FlexiblyIndexed` nodes, the same information as discussed above. This is because from this information we can derive the buffer to dense tensor representation of the local tensor. This can then be used in conjunction with the existing constructs in PyOP2, namely the local-to-global mappings, to perform global assembly just on the non-zeros stored in the buffer.

In particular the parts of PyOP2 that would need to be changed would be the packing and unpacking of `Args` done in the wrapper function. It is these parts that are responsible for extracting the information needed and calling the petsc4py function(s) to add the local contributions to the global matrix (via `MatSetValuesBlockedLocal` or `MatSetValuesLocal` etc.).

Designing a proof of concept

For this project we have not implemented global assembly for sparse local tensors since this would involve large changes in PyOP2 that could not have been done in the time available to us. What we have done instead is completed a proof of concept, to demonstrate how this could be done.

We have hand written and tested the code we want to be generated from PyOP2 for our running example. This is included in listing 5.8. A figure to aid understanding of the code and how PyOP2 could be changed to support sparse tensors is included in 5.6.

The verification that this code is correct and is in fact what we want to see was done through passing in a sparse local tensor, as created in 5.3, and comparing that the resulting global matrix created was the same as the one created from considering the local tensor as dense at each stage (as has been done until now).

```c
void wrap_form00_cell_integral_otherwise(int32_t const start, int32_t const end, Mat
   const mat0, double const *__restrict__ dat0, int32_t const *__restrict__ map0)
{
  ...
  double t1[2 * 3 * 3];  // Sparse local tensor

  for (int32_t n = start; n <= -1 + end; ++n)
  {
    // Local tensor initialised to zero
```

```
 9      for (int32_t i2 = 0; i2 <= 2; ++i2)
10        for (int32_t i3 = 0; i3 <= 1; ++i3)
11          for (int32_t i4 = 0; i4 <= 2; ++i4)
12            t1[6 * i2 + 3 * i3 + i4] = 0.0;
13
14      ...
15      // Inlined local kernel
16      ...
17      for (int32_t form_k0_1 = 0; form_k0_1 <= 2; ++form_k0_1)
18        for (int32_t form_j1 = 0; form_j1 <= 1; ++form_j1)
19          for (int32_t form_j0_1 = 0; form_j0_1 <= 2; ++form_j0_1)
20            t1[3 * form_j0_1 + form_k0_1 + 9 * form_j1] = t1[3 * form_j0_1 + form_k0_1 + 9
     * form_j1] + form_t7[3 * form_j0_1 + form_k0_1];
21
22      // Add local contributions from t1 to global matrix mat0
23      const PetscInt x_indices[] = {2*map0[3 * n], 2*(map0[3 * n]+1), 2*(map0[3 * n]+2)};
24      const PetscInt y_indices[] = {2*map0[3 * n]+1, 2*(map0[3 * n]+1)+1, 2*(map0[3 * n
    ]+2)+1};
25
26      MatSetValuesLocal(mat0, 3, &(x_indices), 3, &(x_indices), &(t1[0]), ADD_VALUES);
27      MatSetValuesLocal(mat0, 3, &(y_indices), 3, &(y_indices), &(t1[9]), ADD_VALUES);
28    }
29  }
```

Listing 5.8: Modified extracts from the desired code generated from PyOP2 wrapper. Comments beginning with // have been inserted for clarity.'...' indicates that there is more content that has not been included here for brevity.

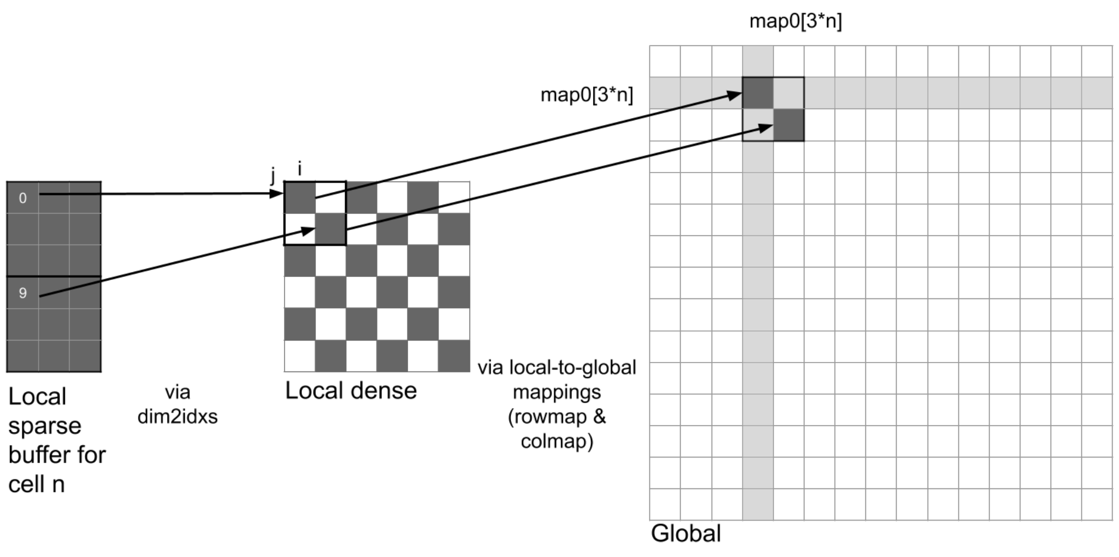

map0[3*n]
map0[3*n]
j
i
0
9
Local sparse buffer for cell n
via dim2idxs
Local dense
via local-to-global mappings (rowmap & colmap)
Global

Figure 5.6: Figure to demonstrate relationship between the buffer, the dense representation of the local tensor and the global matrix for our ongoing example 5.1. The dark grey represents non-zero entries, while the white represents zero entries.

For the code in 5.8, note how the size of temporary t1 storing the local tensor is 2*3*3, exactly the shape of the non-zero buffer of this tensor. Comparing to the code generated before, 5.7, we see that the initialisation of the local tensor and the local kernel operation is much simpler.

Lines 23 and 24 are where we use the information from `dim2idxs` to map from the buffer we have to the dense tensor the assembly process expects. In these lists of indices, we are giving the explicit indexes into the global matrix that we wish to affect. These are the same indices as were affected before, but here they have had a transformation applied to them (*2 for `x_indices`) and (*2 then +1 for `y_indices`) which handles the transformation from buffer to the dense representation of the local tensor.

5.5 Effect

From 5.2.2 we can deduce that, for a 2D tensor, the representation of local tensors as sparse cuts the space and calculation by just under 50%. This is because half the entries in the matrix are zeros, however there is some memory and extra calculation that must be done to handle the mapping from buffer to dense tensor.

By similar reasoning, in 3D memory and calculation are cut by just under 66% since 2/3 of the tensor are zero and accounting for the buffer-dense tensor mapping. At higher dimensions, larger and larger percentages of the tensors are zeros so the reduction in memory and calculation are even larger.

5.6 Evaluation

5.6.1 Limitations

Sparse local tensors cannot be globally assembled or manipulated in Slate Once a local kernel is created, it must either be assembled or have operations performed on it through Slate. We focused on global assembly in PyOP2 since the common use case for most Firedrake users is assembling the local tensor. We found that due to assumptions made in PyOP2 about the structure of the local tensors, adapting this to support sparse tensors would be a big change which we did not have time to do in this project.

Because there is no support for sparse tensors in PyOP2 or Slate, there is nothing that can be done to the sparse local tensors in the local kernel.

Offset unsupported Currently the sparse local tensors only support indexing via indexes and strides, and it is only for tensors indexed with these that we have developed the global assembly for. However, in some cases, the dense local tensor might be indexed with a non-zero offset as well as the indexes and their strides.

Because these offsets are not currently supported, we may get weird behaviour or errors if we attempt to make `StructuredSparseVariables` out of tensors with offsets in their indexes.

We will need to experiment with some examples and develop a way to include offsets in our local buffer representation, as well as modifying the upstream operations that will be done upon the buffer (such as global assembly) to handle the offset.

For more details about this, see future work under 6.2.

5.6.2 Strengths

Sparse local tensors in local kernels We have successfully represented sparse local tensors as buffers with only the non-zeros stored. We have found a way of mapping between the buffer and the dense tensor using `FlexiblyIndexed` nodes.

Proof of concept showing how PyOP2 global assembly can support sparse tensors PyOP2 assumes that there are values for all combinations of the local-to-global degree of freedom maps. We found that this assumption was not correct. This assumption led to the hardcoding of behaviour which is undesireable in the case of sparse local tensors. This is undesireable because it does not allow the needed flexibility to determine how the local data structure is handled.

We developed a solution to show how PyOP2 could be adapted to support sparse tensors. This involved manually writing the code we wanted to be generated from PyOP2, which we verified the correctness of by comparing to the global matrix created from the current method of handling local tensors as dense throughout. We discussed how the information needed in this code could be passed in to PyOP2 through attaching metadata to the local kernel, and that the all the information required can be found on the `FlexiblyIndexed` node we use to create the buffer in the local kernel.

Better understanding of Firedrake and its handling of the structure of things An important contribution of this work is that of discovering more about how the structure of mathematically key constructs interplays across Firedrake systems and the limitations involved. In our example, we looked into the structure of local tensors, particularly in relation to sparsity, and how properties of these could be represented and taken advantage of to improve Firedrake. We found that TSFC, although complex and interlinked, could be adapted quite easily in several ways to support our new structure. PyOP2 on the other hand, was not as easy to adapt to support such things. Here we are touching upon the current limitations of the PyOP2 system.

5.7 Summary

In this chapter we discussed our work on implementing structured sparse tensors in Firedrake.

We successfully implemented sparse local tensors in local kernels, representing the dense tensor by a buffer and using the information in the local tensor's `FlexiblyIndexed` nodes to map between the buffer and the dense tensor.

We demonstrated that the PyOP2 global assembly of structured sparse tensors was possible and discussed how PyOP2 could be adapted to do this. This led to us discovering that the assumption made by PyOP2 is incorrect and that there are not values for all combinations of the local-to-global degree of freedom maps.

We deduce that through implementing sparse local tensors we could reduce the memory and calculation on the local tensors to reduce by over 50% for 2D tensors and over 66% for 3D tensors.

Chapter 6

Conclusion

6.1 Conclusions

The aim of this project was to optimise the layer in Firedrake responsible for tensor computations. This was divided into the following objectives:

1. Reducing the amount of memory needed by temporaries to perform operations on tensors.
2. Taking advantage of the sparsity of local tensors to reduce memory and computation.

6.1.1 Reducing the amount of memory needed by temporaries to perform operations on tensors

Our first step to achieve the above was to implement a new optimisation to reduce the number of temporaries created when performing operations on tensors through smart indexing. This optimisation involves passing through the GEM expression DAG before code generation and eliminating the `ComponentTensor` nodes.

We wanted to understand the specific effects that a reduction in temporaries (as our optimisation does) has in reducing the memory overhead. We developed a model for the amount of memory required by the temporaries to perform operations on tensors. This involved an exploration into the stacking effects of different operations, and analysis of the effects of the choice of finite element family, mesh. etc.

We found that our optimisation still created some unneccessary memory associated with temporaries, but it did reduce the amount of memory needed by 55% - 65%, with the representative hybridization example discussed in 4.2.6 having a reduction by 60%. Thus to a greater extent we have achieved our first objective.

6.1.2 Taking advantage of the sparsity of local tensors to reduce memory and computation.

We successfully implemented sparse local tensors in local kernels, representing the tensors as a buffer of the non-zero values and a mapping from buffer to the original dense tensor, which was derived from the information in the tensor's `FlexiblyIndexed` node.

Unfortunately, we found that due to an assumption made in PyOP2 about the structure of the local tensors, adapting this to support sparse tensors would be a big change, and we didn't have time to do this during this project. PyOP2 assumes that there are values for all combinations of the local-to-global degree of freedom maps, hence all local data strutures are hardcoded to be treated as dense. This means there is no flexibility for us to define a different way in which PyOP2 should handle the data.

Instead, we developed a solution to show how PyOP2 could be adapted to support sparse tensors. This involved manually writing the code we wanted to be generated from PyOP2. We verified the correctness of this code by comparing it's global matrix to the actual global matrix which we got from running the non-sparse version of the code. Developing a solution involved deducing the information that would need to be passed to PyOP2 and suggestions on what parts of the code need to be changed.

Although we have not been able to change Firedrake such that local tensor sparsity can be taken advantage of, we have taken a significant step towards achieving this. We have also deduced that by completing the implementation of sparse local tensors we could reduce the memory and calculation on the local tensors to by over 50% for 2D tensors and over 66% for 3D tensors. This would further advance achieving our first objective.

6.2 Future work

Further reductions in memory required for temporaries More can be done to reduce the unneccessary memory for temporaries. An example of this is discussed in 4.2.1. We have not come across many cases of unneccessary temporary creation still in the system, and the example given in 4.2.1 would yield a very small reduction in memory. Hence, it is most likely not a significant change.

Another way we can reduce the memory required is by finding a way to stop needing to recalculate differently indexed expressions. An example where we have seen this occur is the recalculation of `-B` in `-B*-A*-A*-B` (see 4.2.1).

Further investigation into stacking effects More can be done to investigate the stacking effects of operations as discussed in our model. Even if there are stacking effects unaccounted for in our model, it is likely that the number and shape of temporaries is not affected very significantly. Therefore, such stacking effects may cause the model to slightly over or under predict. For the foreseeable use cases of this model, inaccuracies in the prediction that are this small are acceptable.

Extend the model to handle more possible cases Our model currently includes only common choices for things like mesh and finite element, as this was sufficient for our use cases. There are other values of these that the model has not been built or tested with. Thus extending the model to predict the memory used for these cases might be of interest to some users.

Automation of making predictions using the model for memory needed for temporaries The automation of the model using a computer program may be beneficial, depending on how the model is used. This would avoid the tediousness of making predictions manually, especially for long or complex sets of operations.

Support for sparse local tensors in PyOP2 Adding support for structurally sparse local tensors into PyOP2 would allow sparse tensors in local kernels to be assembled into global tensors, hence allowing useful work to be done with them. Since most users of Firedrake do global assembly rather than going through Slate, this is where the initial efforts would be most effective.

For details on how this could be achieved see 5.4.4.

Support for sparse local tensors in Slate Adding support for working with structured sparse local tensors in Slate would mean the usual Slate operations could then be performed on these local tensors. This would give the benefits of Slate with the reduced memory and calculation from using these sparse tensors.

Adding support for non-zero offsets in sparse local tensors Currently, the sparse local tensor and global assembly solutions support only indexing via indexes and strides. They do not support indexing with a non-zero offset.

To support non-zero offsets we would need to develop a way to include offsets in the local buffer representation, as well as modifying the upstream operations that will be done upon the buffer (such as global assembly) to handle the offset.

One way of implementing offsets is to add another dimension in the buffer, whose extent would be the number of unique offsets used. Information could then be passed from the local kernel to PyOP2 about the mapping between the offset value and its index in the offset dimension.

Appendix A

Notation, Acronyms and Glossary

A.1 Notation

n (.,.) OR n (.) OR n ()	*Used when discussing the temporaries needed as a part of performing operations on tensor. n is the number of temporaries of this shape, (.,.) denotes a matrix temporary with a shape that is a 2-tuple, (.) denotes a vector temporary with a 1-tuple shape, () denotes a scalar temporary, i.e. just a number.*

A.2 Acronyms and Shorthands

DAG	*Directed acyclic graph*
DSL	*Domain Specific Language*
FEM	*Finite Element Method*
Flops	*Floating-point operations*
mat-mat	*Matrix-matrix multiplication*
mat-vec	*Matrix-vector multiplication*
PDE	*Partial Differential Equation*
UFL	*Unified Form Language*

A.3 Glossary

Dense linear algebra	*Involves matrices with mostly non-zero values. This is in comparision to sparse linear algebra.*
DSL	*Domain Specific Language. A language specialized for a particular field of study.*
Facet	*A feature of a geometric structure, usually of one dimension fewer than the structure itself. For example, for a polyhedron a facet is any polygon whose corners are vertices of the polyhedron and that is not a face. [59]*
Flux	*The (perceived) flow of something over or through something.*
Geometry	*Geometry (e.g. of a mesh) is characterized by the coordinates describing where the parts of a structure (e.g. vertices) are in space. [60]*
Kernel	*A single function call.*
Mesh	*The partioning of a domain into polytopes.*
Mesh entities	*Local mesh entities of a triangle are the vertices, the edges and the triangle itself. Global mesh entities of a mesh are all the vertices, edges and triangles in the mesh.*
Multilinear	*An object in multiple variables is 'multilinear' if it is linear in all its variables separately.*
OP2	*A library for writing parallel programs to do computations on unstructured meshes. [3]*
Polytope	*A geometric object with "flat" sides. This is an n-dimensional generalisation of a 2-dimensional polygon and a 3-dimensional polyhedron*
Preconditioner	*Preconditioning means applying a transformation, called the preconditioner, that changes a problem into a more appropriate form for solving using numerical methods. A more appropriate form usually means a form that is equivalent to the original system but which converges quicker for iterative methods.*
Stacking effects	*When several operations on tensors are performed in a sequence which allows temporaries to be re-used beween operations. This can result in a smaller number of temporaries, and sometimes in different shaped temporaries, than would be expected from treating the operations in isolation.*
Tensor	*An n-dimensional container for data, e.g. matrices (2D tensors), vectors (1D tensors), scalars (0D tensor). Tensors also describe linear transformations between tensors, e.g. dot product (maps two vectors to a scalar).*
Topology	*Topology (e.g. of a mesh) includes the parts of a structure (e.g. cells, edges, vertices) and the adjacency relationships between them (e.g. vertices to edges). [60]*
UFL	*Unified Form Language. A DSL for representing weak forms of PDEs.*

Appendix B

Specification of testing hardware

```
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              12
On-line CPU(s) list: 0-11
Thread(s) per core:  2
Core(s) per socket:  6
Socket(s):           1
NUMA node(s):        1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               158
Model name:          Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz
Stepping:            10
CPU MHz:             800.030
CPU max MHz:         4100.0000
CPU min MHz:         800.0000
BogoMIPS:            4416.00
Virtualisation:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            9216K
NUMA node0 CPU(s):   0-11
Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat
 pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm
constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid
aperfmperf tsc_known_freq pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg
fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave
 avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb invpcid_single pti ssbd
ibrs ibpb stibp tpr_shadow vnmi flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1
avx2 smep bmi2 erms invpcid mpx rdseed adx smap clflushopt intel_pt xsaveopt xsavec
xgetbv1 xsaves dtherm ida arat pln pts hwp hwp_notify hwp_act_window hwp_epp md_clear
flush_l1d
```

Listing B.1: Basic hardware specification of computer used for experiments

Appendix C

Code before and after removing unneccessary temporaries

Code before and after removal of unneccessary temporaries. Unless otherwise specified, the expressions on which the following operations are performed are those from the discontinous Helmholtz equation on cell integrals:

$$
\begin{aligned}
a &= \int \nabla v \cdot \nabla u + vu \; dx \\
L &= \int fv \; dx \\
f &= \left(1 + 8\pi^2\right) \cos\left(2\pi x\right) \cos\left(2\pi y\right)
\end{aligned}
\tag{C.1}
$$

C.1 AssembledVector

```
for i0
  t0[i0] = VecTemp0[i0]  {id=insn, priority=2}
end i0
for i
  output[i] = output[i] + t0[i]  {id=insn_0, priority=1}
end i
```

Listing C.1: Original `assemble(AssembledVector(Function(assemble(L))))`

```
for i
  output[i] = output[i] + VecTemp0[i]  {id=insn, priority=1}
end i
```

Listing C.2: Optimised `assemble(AssembledVector(Function(assemble(L))))`

C.2 Negative

For the arithmetic negation of a (i.e. -a), this is the original code:

```
for i1, i0
    t0[i0, i1] = T0[i0, i1]  {id=insn, priority=3}
end i1, i0
for i6, i5
    t1[i5, i6] = (-1.0)*t0[i5, i6]  {id=insn_0, priority=2}
end i6, i5
for i, i_0
    output[i, i_0] = output[i, i_0] + t1[i, i_0]  {id=insn_1, priority=1}
end i, i_0
```

Listing C.3: Original `assemble(-Tensor(a))`

We can see we produce t0 and t1 as unneccessary temporaries. We again remove the Component-Tensor nodes and associated Indexed nodes and this yields the below code:

```
for i_0, i
    output[i, i_0] = output[i, i_0] + (-1.0)*T0[i, i_0]  {id=insn, priority=1}
end i_0, i
```

Listing C.4: Optimised `assemble(-Tensor(a))`

This outcome also applies to stacked negations, with the optimised code output being only one line of code with one necessary temporary.

C.3 Inverse local

```
for i0, i1
    t0[i0, i1] = T0[i0, i1]  {id=insn, priority=4}
end i0, i1
for i5, i6
    t1[i5, i6] = t0[i5, i6]  {id=insn_0, priority=3}
end i5, i6
SubArrayRef((i7, i8), (t1[i7, i8])) = inv(SubArrayRef((i5, i6), (t1[i5, i6])))  {id=insn_1
    , priority=2}
for i_0, i
    output[i, i_0] = output[i, i_0] + t1[i, i_0]  {id=insn_2, priority=1}
end i_0, i
```

Listing C.5: Original `Tensor(a).inv`

```
for i6, i5
    t0[i5, i6] = T0[i5, i6]  {id=insn, priority=3}
end i6, i5
SubArrayRef((i7, i8), (t0[i7, i8])) = inv(SubArrayRef((i5, i6), (t0[i5, i6])))  {id=insn_0
    , priority=2}
for i, i_0
    output[i, i_0] = output[i, i_0] + t0[i, i_0]  {id=insn_1, priority=1}
end i, i_0
```

Listing C.6: Optimised `Tensor(a).inv`

C.4 Addition

```
for i0, i1
    t0[i0, i1] = T0[i0, i1]  {id=insn, priority=3}
end i0, i1
for i6, i5
    t1[i5, i6] = t0[i5, i6] + t0[i5, i6]  {id=insn_0, priority=2}
end i6, i5
for i, i_0
    output[i, i_0] = output[i, i_0] + t1[i, i_0]  {id=insn_1, priority=1}
end i, i_0
```

Listing C.7: Original `assemble(Tensor(a) + Tensor(a))`

```
for i, i_0
    output[i, i_0] = output[i, i_0] + T0[i, i_0] + T0[i, i_0]  {id=insn, priority=1}
```

```
end i, i_0
```

Listing C.8: Optimised `assemble(Tensor(a) + Tensor(a))`

C.5 Multiplication

C.5.1 Matrix-Vector multiplication

```
for i1, i2
    t0[i1, i2] = T0[i1, i2]  {id=insn, priority=6}
end i1, i2
for i0
  t1[i0] = VecTemp0[i0]  {id=insn_0, priority=5}
end i0
for i
  t2[i] = 0.0  {id=insn_1, priority=4}
  for i_0
    t2[i] = t2[i] + t0[i, i_0]*t1[i_0]  {id=insn_2, priority=3}
end i, i_0
for i6
  t3[i6] = t2[i6]  {id=insn_3, priority=2}
end i6
for i_1
  output[i_1] = output[i_1] + t3[i_1]  {id=insn_4, priority=1}
end i_1
```

Listing C.9: Original `Tensor(a) * AssembledVector(Function(assemble(L)))`

```
for i_0, i
    output[i] = output[i] + T0[i, i_0]*VecTemp0[i_0]  {id=insn, priority=1}
end i_0, i
```

Listing C.10: Optimised code for `Tensor(a) * AssembledVector(Function(assemble(L)))`

C.5.2 Matrix-matrix

Here we multiplied a together with $a2 = \int \nabla v \cdot \nabla u \, dx$

```
for i2, i3
    t0[i2, i3] = T1[i2, i3]  {id=insn, priority=6}
end i2, i3
for i0, i1
    t1[i0, i1] = T0[i0, i1]  {id=insn_0, priority=5}
end i0, i1
for i, i_0
    t2[i, i_0] = 0.0  {id=insn_1, priority=4}
  end i_0
  for i_1, i_2
      t2[i, i_2] = t2[i, i_2] + t0[i, i_1]*t1[i_1, i_2]  {id=insn_2, priority=3}
end i, i_1, i_2
for i12, i9
    t3[i9, i12] = t2[i9, i12]  {id=insn_3, priority=2}
end i12, i9
for i_4, i_3
    output[i_3, i_4] = output[i_3, i_4] + t3[i_3, i_4]  {id=insn_4, priority=1}
end i_4, i_3
```

Listing C.11: Original `Tensor(a)*Tensor(a2)`

```
for i_0, i, i_1
    output[i, i_1] = output[i, i_1] + T1[i, i_0]*T0[i_0, i_1]  {id=insn, priority=1}
end i_0, i, i_1
```

Listing C.12: Optimised code for `Tensor(a)*Tensor(a2)`

C.6 Solve

For the below example, the UFL setup code is as follows:

```
_A = Tensor(a)
A = assemble(A)
_F = AssembledVector(assemble(L))
F = assemble(_F)
u = Function(V)
```

C.6.1 Global solve: solve(A, u, F, solver_parameters={'ksp_type':'cg'})

```
for i2, i1
    t0[i1, i2] = T0[i1, i2]  {id=insn, priority=6}
end i2, i1
for i0
  t1[i0] = VecTemp0[i0]  {id=insn_0, priority=5}
end i0
for i
  t2[i] = 0.0  {id=insn_1, priority=4}
  for i_0
    t2[i] = t2[i] + t0[i, i_0]*t1[i_0]  {id=insn_2, priority=3}
end i, i_0
for i6
  t3[i6] = t2[i6]  {id=insn_3, priority=2}
end i6
for i_1
  output[i_1] = output[i_1] + t3[i_1]  {id=insn_4, priority=1}
end i_1
```

Listing C.13: Original output code for global solve

```
for i_0, i
    output[i] = output[i] + T0[i, i_0]*VecTemp0[i_0]  {id=insn, priority=1}
end i_0, i
```

Listing C.14: Optimised output code for global solve

C.6.2 Local solve: assemble(_A.solve(_F))

```
for i1, i2
    t0[i1, i2] = T0[i1, i2]  {id=insn, priority=8}
end i1, i2
for i7, i6
    t1[i6, i7] = t0[i6, i7]  {id=insn_0, priority=7}
end i7, i6
SubArrayRef((i8, i9), (t1[i8, i9])) = inv(SubArrayRef((i6, i7), (t1[i6, i7])))  {id=insn_1
    , priority=6}
for i0
  t3[i0] = VecTemp0[i0]  {id=insn_2, priority=5}
end i0
```

```
for i
  t4[i] = 0.0  {id=insn_3, priority=4}
  for i_0
    t4[i] = t4[i] + t1[i, i_0]*t3[i_0]  {id=insn_4, priority=3}
end i, i_0
for i10
  t5[i10] = t4[i10]  {id=insn_5, priority=2}
end i10
for i_1
  output[i_1] = output[i_1] + t5[i_1]  {id=insn_6, priority=1}
end i_1
```

Listing C.15: Original `assemble(Tensor(a).solve(AssembledVector(assemble(L))))`

```
for i6, i7
    t0[i6, i7] = T0[i6, i7]  {id=insn, priority=3}
end i6, i7
SubArrayRef((i8, i9), (t0[i8, i9])) = inv(SubArrayRef((i6, i7), (t0[i6, i7])))  {id=insn_0
    , priority=2}
for i_0, i
    output[i] = output[i] + t0[i, i_0]*VecTemp0[i_0]  {id=insn_1, priority=1}
end i_0, i
```

Listing C.16: Optimised `assemble(Tensor(a).solve(AssembledVector(assemble(L))))`

C.7 Blocks

No unneccessary temporaries for this operation. Here we let $A = Tensor(inner(u, w) * dx + p * q * dx - div(w) * p * dx + q * div(u) * dx)$.

```
for i, i_0
    output[i, i_0] = output[i, i_0] + T0[i, 3 + i_0]  {id=insn, priority=1}
end i, i_0
```

Listing C.17: Original and optimised code for `A.blocks[0,1]`

```
for i, i_0
    output[i, i_0] = output[i, i_0] + T0[3 + i, 3 + i_0]  {id=insn, priority=1}
end i, i_0
```

Listing C.18: Original and optimised code for `A.blocks[1,1]`

C.8 Hybridization

```
# Temporaries and code for Schur right-hand side
TEMPORARIES:
T0: type: np:dtype('float64'), shape: (4, 4), dim_tags: (N1:stride:4, N0:stride:1) scope:
    local
T1: type: np:dtype('float64'), shape: (3, 4), dim_tags: (N1:stride:4, N0:stride:1) scope:
    local
t0: type: np:dtype('float64'), shape: (4, 4), dim_tags: (N1:stride:4, N0:stride:1) scope:
    local
t1: type: np:dtype('float64'), shape: (4, 4), dim_tags: (N1:stride:4, N0:stride:1) scope:
    local
t3: type: np:dtype('float64'), shape: (3, 4), dim_tags: (N1:stride:4, N0:stride:1) scope:
    local
t4: type: np:dtype('float64'), shape: (3, 4), dim_tags: (N1:stride:4, N0:stride:1) scope:
    local
```

```
t5: type: np:dtype('float64'), shape: (3, 4), dim_tags: (N1:stride:4, N0:stride:1) scope:
    local
t6: type: np:dtype('float64'), shape: (4), dim_tags: (N0:stride:1) scope:local
t7: type: np:dtype('float64'), shape: (3), dim_tags: (N0:stride:1) scope:local
t8: type: np:dtype('float64'), shape: (3), dim_tags: (N0:stride:1) scope:local
---------------------------------------------------------------------------
INSTRUCTIONS:
for i2, i1
    t0[i1, i2] = T0[i1, i2]  {id=insn, priority=12}
end i2, i1
for i20, i19
    t1[i19, i20] = t0[i19, i20]  {id=insn_0, priority=11}
end i20, i19
SubArrayRef((i21, i22), (t1[i21, i22])) = inv(SubArrayRef((i19, i20), (t1[i19, i20])))  {
    id=insn_1, priority=10}
for i3, i4
    t3[i3, i4] = T1[i3, i4]  {id=insn_2, priority=9}
end i3, i4
for i, i_0
    t4[i, i_0] = 0.0  {id=insn_3, priority=8}
  end i_0
  for i_1, i_2
      t4[i, i_2] = t4[i, i_2] + t3[i, i_1]*t1[i_1, i_2]  {id=insn_4, priority=7}
end i, i_1, i_2
for i23, i26
    t5[i23, i26] = t4[i23, i26]  {id=insn_5, priority=6}
end i23, i26
for i18
  t6[i18] = VecTemp0[i18]  {id=insn_6, priority=5}
end i18
for i_3
  t7[i_3] = 0.0  {id=insn_7, priority=4}
  for i_4
    t7[i_3] = t7[i_3] + t5[i_3, i_4]*t6[i_4]  {id=insn_8, priority=3}
end i_3, i_4
for i27
  t8[i27] = t7[i27]  {id=insn_9, priority=2}
end i27
for i_5
  output[i_5] = output[i_5] + t8[i_5]  {id=insn_10, priority=1}
end i_5

# Temporaries and code for Schur-complement
TEMPORARIES:
T0: type: np:dtype('float64'), shape: (3, 4), dim_tags: (N1:stride:4, N0:stride:1) scope:
    local
T1: type: np:dtype('float64'), shape: (4, 4), dim_tags: (N1:stride:4, N0:stride:1) scope:
    local
t0: type: np:dtype('float64'), shape: (4, 4), dim_tags: (N1:stride:4, N0:stride:1) scope:
    local
t1: type: np:dtype('float64'), shape: (3, 4), dim_tags: (N1:stride:4, N0:stride:1) scope:
    local
t2: type: np:dtype('float64'), shape: (4, 4), dim_tags: (N1:stride:4, N0:stride:1) scope:
    local
t4: type: np:dtype('float64'), shape: (3, 4), dim_tags: (N1:stride:4, N0:stride:1) scope:
    local
t5: type: np:dtype('float64'), shape: (3, 4), dim_tags: (N1:stride:4, N0:stride:1) scope:
    local
t6: type: np:dtype('float64'), shape: (4, 3), dim_tags: (N1:stride:3, N0:stride:1) scope:
    local
t7: type: np:dtype('float64'), shape: (3, 3), dim_tags: (N1:stride:3, N0:stride:1) scope:
    local
```

```
t8: type: np:dtype('float64'), shape: (3, 3), dim_tags: (N1:stride:3, N0:stride:1) scope:
    local
---------------------------------------------------------------------------
INSTRUCTIONS:
for i2, i3
    t0[i2, i3] = T1[i2, i3]  {id=insn, priority=12}
end i2, i3
for i1, i0
    t1[i0, i1] = T0[i0, i1]  {id=insn_0, priority=11}
end i1, i0
for i18, i17
    t2[i17, i18] = t0[i17, i18]  {id=insn_1, priority=10}
end i18, i17
SubArrayRef((i19, i20), (t2[i19, i20])) = inv(SubArrayRef((i17, i18), (t2[i17, i18])))  {
    id=insn_2, priority=9}
for i, i_0
    t4[i, i_0] = 0.0  {id=insn_3, priority=8}
  end i_0
  for i_1, i_2
      t4[i, i_2] = t4[i, i_2] + t1[i, i_1]*t2[i_1, i_2]  {id=insn_4, priority=7}
end i, i_1, i_2
for i24, i21
    t5[i21, i24] = t4[i21, i24]  {id=insn_5, priority=6}
end i24, i21
for i26, i25
    t6[i26, i25] = t1[i25, i26]  {id=insn_6, priority=5}
end i26, i25
for i_3, i_4
    t7[i_3, i_4] = 0.0  {id=insn_7, priority=4}
  end i_4
  for i_6, i_5
      t7[i_3, i_6] = t7[i_3, i_6] + t5[i_3, i_5]*t6[i_5, i_6]  {id=insn_8, priority=3}
end i_3, i_6, i_5
for i27, i30
    t8[i27, i30] = t7[i27, i30]  {id=insn_9, priority=2}
end i27, i30
for i_8, i_7
    output[i_7, i_8] = output[i_7, i_8] + t8[i_7, i_8]  {id=insn_10, priority=1}
end i_8, i_7
```

Listing C.19: Modified extract of original output for hybridization example.

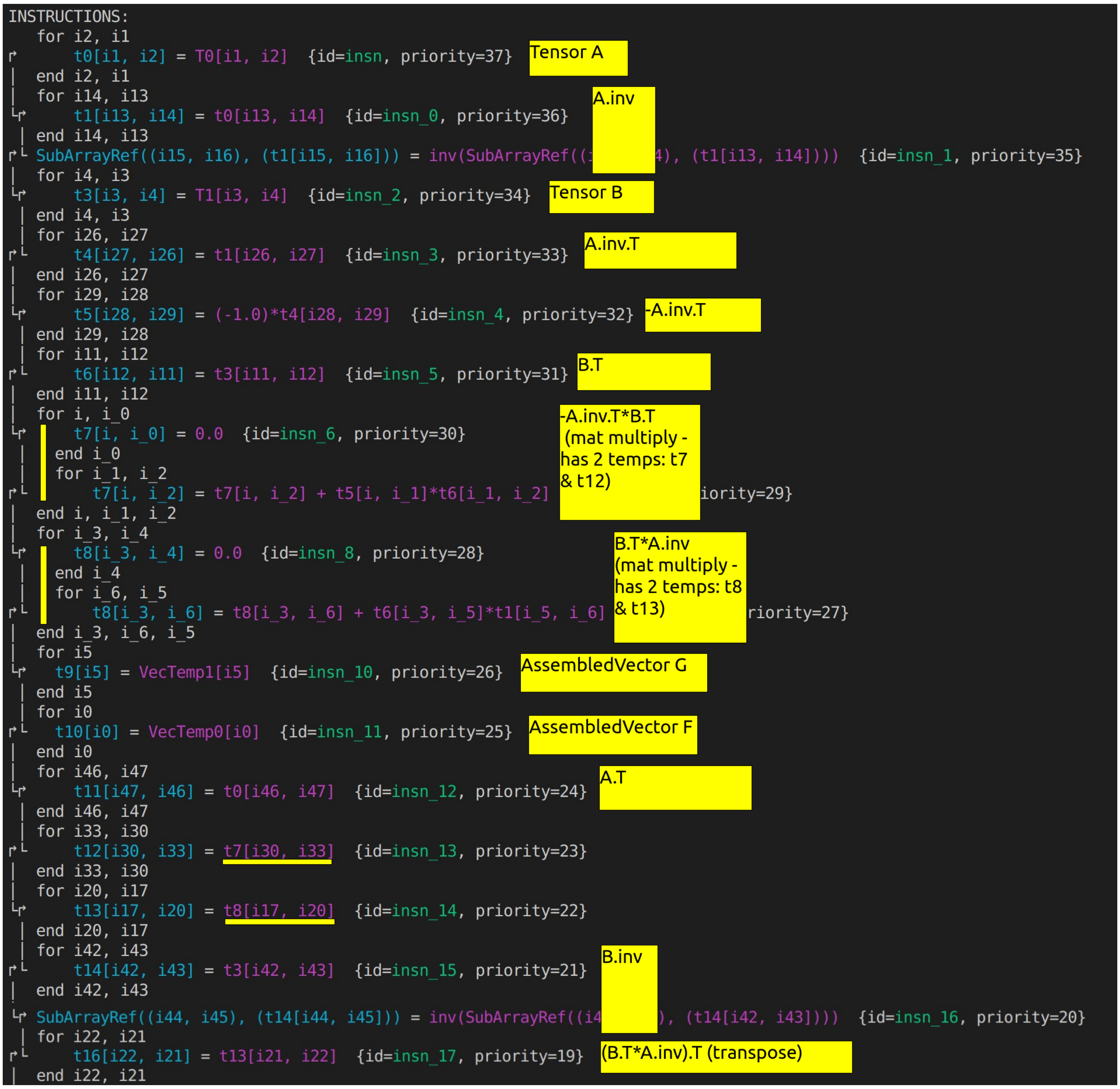
```
# Temporaries and code for Schur right-hand side
TEMPORARIES:
T0: type: np:dtype('float64'), shape: (4, 4), dim_tags: (N1:stride:4, N0:stride:1) scope:
    local
T1: type: np:dtype('float64'), shape: (3, 4), dim_tags: (N1:stride:4, N0:stride:1) scope:
    local
t0: type: np:dtype('float64'), shape: (4, 4), dim_tags: (N1:stride:4, N0:stride:1) scope:
    local
t2: type: np:dtype('float64'), shape: (4), dim_tags: (N0:stride:1) scope:local
---------------------------------------------------------------------------
INSTRUCTIONS:
for i20, i19
    t0[i19, i20] = T0[i19, i20]  {id=insn, priority=5}
end i20, i19
SubArrayRef((i21, i22), (t0[i21, i22])) = inv(SubArrayRef((i19, i20), (t0[i19, i20])))  {
    id=insn_0, priority=4}
for i, i_0
    t2[i_0] = 0.0  {id=insn_1, priority=3}
  end i_0
  for i_2, i_1
      t2[i_2] = t2[i_2] + T1[i, i_1]*t0[i_1, i_2]  {id=insn_2, priority=2}
  end i_2, i_1
  for i_3
    output[i] = output[i] + t2[i_3]*VecTemp0[i_3]  {id=insn_3, priority=1}
end i, i_3


# Temporaries and code for Schur-complement
TEMPORARIES:
T0: type: np:dtype('float64'), shape: (3, 4), dim_tags: (N1:stride:4, N0:stride:1) scope:
    local
T1: type: np:dtype('float64'), shape: (4, 4), dim_tags: (N1:stride:4, N0:stride:1) scope:
    local
t0: type: np:dtype('float64'), shape: (4, 4), dim_tags: (N1:stride:4, N0:stride:1) scope:
    local
t2: type: np:dtype('float64'), shape: (4), dim_tags: (N0:stride:1) scope:local
---------------------------------------------------------------------------
INSTRUCTIONS:
for i18, i17
    t0[i17, i18] = T1[i17, i18]  {id=insn, priority=5}
end i18, i17
SubArrayRef((i19, i20), (t0[i19, i20])) = inv(SubArrayRef((i17, i18), (t0[i17, i18])))  {
    id=insn_0, priority=4}
for i, i_0
    t2[i_0] = 0.0  {id=insn_1, priority=3}
  end i_0
  for i_2, i_1
      t2[i_2] = t2[i_2] + T0[i, i_1]*t0[i_1, i_2]  {id=insn_2, priority=2}
  end i_2, i_1
  for i_3, i_4
      output[i, i_4] = output[i, i_4] + t2[i_3]*T0[i_4, i_3]  {id=insn_3, priority=1}
end i, i_3, i_4
```

Listing C.20: Modified extract of optimised output for hybridization example.

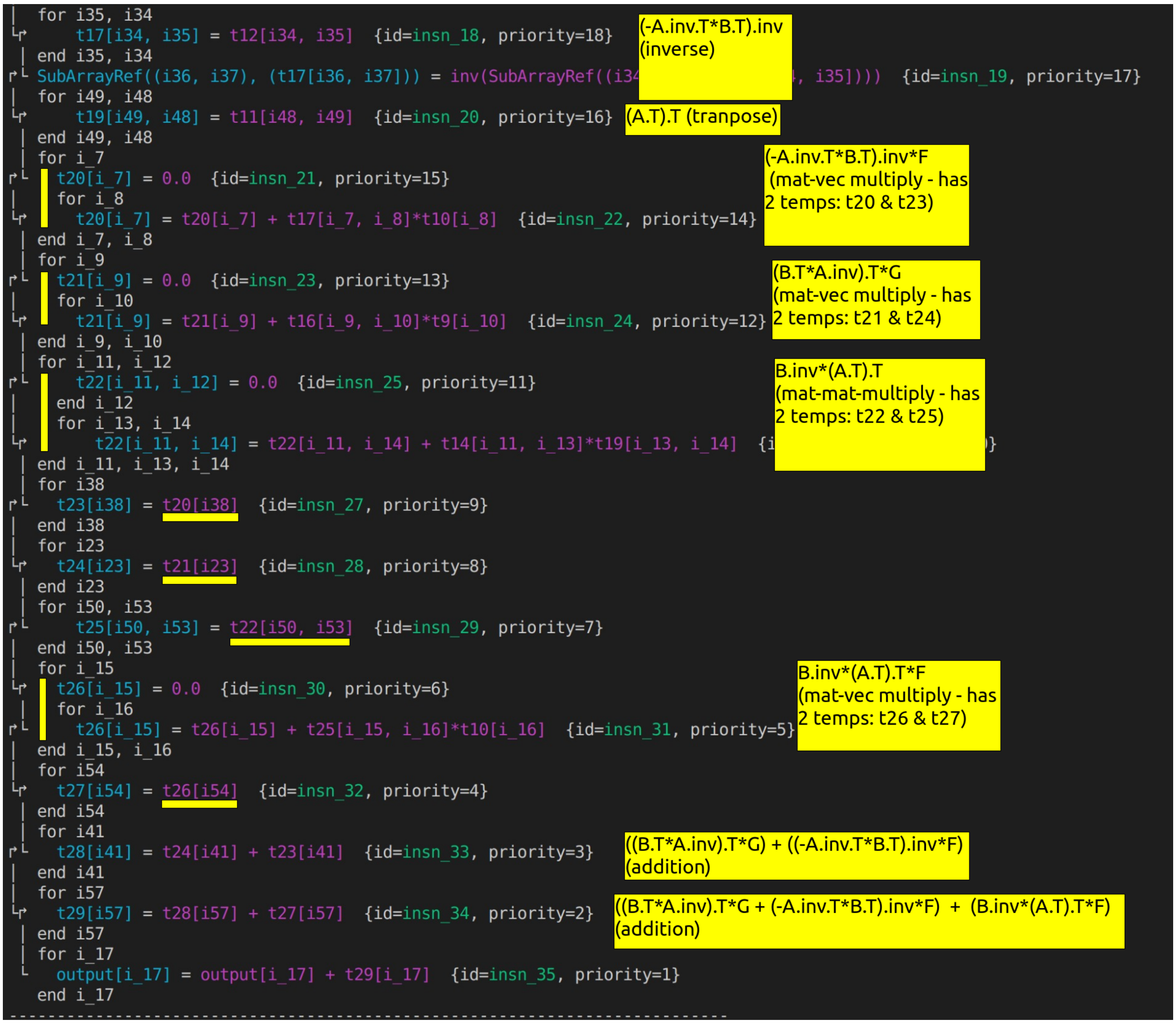
Appendix D

Aggressive unary op nesting output

D.1 Before optimisation


Tensor A
A.inv
Tensor B
A.inv.T
-A.inv.T
B.T
-A.inv.T*B.T (mat multiply - has 2 temps: t7 & t12)
B.T*A.inv (mat multiply - has 2 temps: t8 & t13)
AssembledVector G
AssembledVector F
A.T
B.inv
(B.T*A.inv).T (transpose)

```
INSTRUCTIONS:
   for i2, i1
↱      t0[i1, i2] = T0[i1, i2]  {id=insn, priority=37}
│  end i2, i1
│  for i14, i13
└↱     t1[i13, i14] = t0[i13, i14]  {id=insn_0, priority=36}
 │ end i14, i13
↱└ SubArrayRef((i15, i16), (t1[i15, i16])) = inv(SubArrayRef((i        4), (t1[i13, i14])))  {id=insn_1, priority=35}
│  for i4, i3
└↱     t3[i3, i4] = T1[i3, i4]  {id=insn_2, priority=34}
 │ end i4, i3
 │ for i26, i27
↱└     t4[i27, i26] = t1[i26, i27]  {id=insn_3, priority=33}
│  end i26, i27
│  for i29, i28
└↱     t5[i28, i29] = (-1.0)*t4[i28, i29]  {id=insn_4, priority=32}
 │ end i29, i28
 │ for i11, i12
↱└     t6[i12, i11] = t3[i11, i12]  {id=insn_5, priority=31}
│  end i11, i12
│  for i, i_0
└↱ │   t7[i, i_0] = 0.0  {id=insn_6, priority=30}
 │ │  end i_0
 │ │  for i_1, i_2
↱└ │     t7[i, i_2] = t7[i, i_2] + t5[i, i_1]*t6[i_1, i_2]            iority=29}
│  end i, i_1, i_2
│  for i_3, i_4
└↱ │   t8[i_3, i_4] = 0.0  {id=insn_8, priority=28}
 │ │  end i_4
 │ │  for i_6, i_5
↱└ │     t8[i_3, i_6] = t8[i_3, i_6] + t6[i_3, i_5]*t1[i_5, i_6]            riority=27}
│  end i_3, i_6, i_5
│  for i5
└↱    t9[i5] = VecTemp1[i5]  {id=insn_10, priority=26}
 │ end i5
 │ for i0
↱└    t10[i0] = VecTemp0[i0]  {id=insn_11, priority=25}
│  end i0
│  for i46, i47
└↱     t11[i47, i46] = t0[i46, i47]  {id=insn_12, priority=24}
 │ end i46, i47
 │ for i33, i30
↱└     t12[i30, i33] = t7[i30, i33]  {id=insn_13, priority=23}
│  end i33, i30
│  for i20, i17
└↱     t13[i17, i20] = t8[i17, i20]  {id=insn_14, priority=22}
 │ end i20, i17
 │ for i42, i43
↱└     t14[i42, i43] = t3[i42, i43]  {id=insn_15, priority=21}
│  end i42, i43
└↱ SubArrayRef((i44, i45), (t14[i44, i45])) = inv(SubArrayRef((i4        ), (t14[i42, i43])))  {id=insn_16, priority=20}
 │ for i22, i21
↱└     t16[i22, i21] = t13[i21, i22]  {id=insn_17, priority=19}
│  end i22, i21
```



```
│   for i35, i34
└↱      t17[i34, i35] = t12[i34, i35]  {id=insn_18, priority=18}
 │ end i35, i34
↱└ SubArrayRef((i36, i37), (t17[i36, i37])) = inv(SubArrayRef((i34                4, i35])))  {id=insn_19, priority=17}
│   for i49, i48
└↱      t19[i49, i48] = t11[i48, i49]  {id=insn_20, priority=16}
 │ end i49, i48
 │ for i_7
↱└   t20[i_7] = 0.0  {id=insn_21, priority=15}
│     for i_8
└↱      t20[i_7] = t20[i_7] + t17[i_7, i_8]*t10[i_8]  {id=insn_22, priority=14}
 │ end i_7, i_8
 │ for i_9
↱└   t21[i_9] = 0.0  {id=insn_23, priority=13}
│     for i_10
└↱      t21[i_9] = t21[i_9] + t16[i_9, i_10]*t9[i_10]  {id=insn_24, priority=12}
 │ end i_9, i_10
 │ for i_11, i_12
↱└      t22[i_11, i_12] = 0.0  {id=insn_25, priority=11}
│     end i_12
│     for i_13, i_14
└↱        t22[i_11, i_14] = t22[i_11, i_14] + t14[i_11, i_13]*t19[i_13, i_14]  {i                          }
 │ end i_11, i_13, i_14
 │ for i38
↱└   t23[i38] = t20[i38]  {id=insn_27, priority=9}
│   end i38
│   for i23
└↱   t24[i23] = t21[i23]  {id=insn_28, priority=8}
 │ end i23
 │ for i50, i53
↱└      t25[i50, i53] = t22[i50, i53]  {id=insn_29, priority=7}
│   end i50, i53
│   for i_15
└↱   t26[i_15] = 0.0  {id=insn_30, priority=6}
│     for i_16
↱└      t26[i_15] = t26[i_15] + t25[i_15, i_16]*t10[i_16]  {id=insn_31, priority=5}
│   end i_15, i_16
│   for i54
└↱   t27[i54] = t26[i54]  {id=insn_32, priority=4}
 │ end i54
 │ for i41
↱└   t28[i41] = t24[i41] + t23[i41]  {id=insn_33, priority=3}
│   end i41
│   for i57
└↱   t29[i57] = t28[i57] + t27[i57]  {id=insn_34, priority=2}
 │ end i57
 │ for i_17
 └   output[i_17] = output[i_17] + t29[i_17]  {id=insn_35, priority=1}
    end i_17
---------------------------------------------------------------------------
```

(-A.inv.T*B.T).inv
(inverse)

(A.T).T (tranpose)

(-A.inv.T*B.T).inv*F
(mat-vec multiply - has
2 temps: t20 & t23)

(B.T*A.inv).T*G
(mat-vec multiply - has
2 temps: t21 & t24)

B.inv*(A.T).T
(mat-mat-multiply - has
2 temps: t22 & t25)

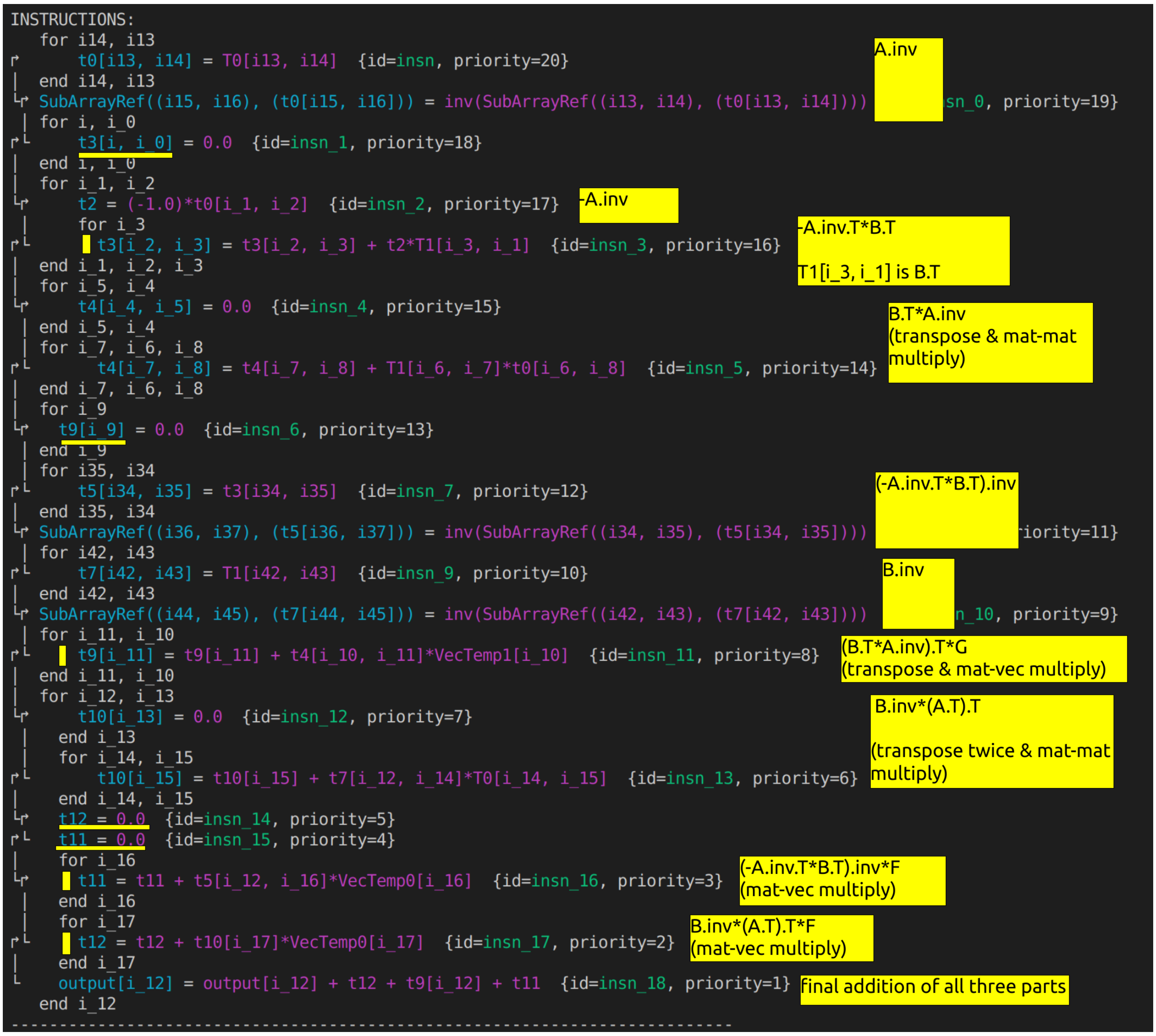
B.inv*(A.T).T*F
(mat-vec multiply - has
2 temps: t26 & t27)

((B.T*A.inv).T*G) + ((-A.inv.T*B.T).inv*F)
(addition)

((B.T*A.inv).T*G + (-A.inv.T*B.T).inv*F) + (B.inv*(A.T).T*F)
(addition)

Figure D.1: Annotated output from aggressive unary op nesting example, before optimisation

D.2 After optimisation



```
INSTRUCTIONS:
   for i14, i13
↱      t0[i13, i14] = T0[i13, i14]  {id=insn, priority=20}
│  end i14, i13
└↱ SubArrayRef((i15, i16), (t0[i15, i16])) = inv(SubArrayRef((i13, i14), (t0[i13, i14])))        sn_0, priority=19}
 │ for i, i_0
↱└     t3[i, i_0] = 0.0  {id=insn_1, priority=18}
│  end i, i_0
│  for i_1, i_2
└↱     t2 = (-1.0)*t0[i_1, i_2]  {id=insn_2, priority=17}
 │     for i_3
↱└       t3[i_2, i_3] = t3[i_2, i_3] + t2*T1[i_3, i_1]  {id=insn_3, priority=16}
│  end i_1, i_2, i_3
│  for i_5, i_4
└↱     t4[i_4, i_5] = 0.0  {id=insn_4, priority=15}
 │ end i_5, i_4
 │ for i_7, i_6, i_8
↱└       t4[i_7, i_8] = t4[i_7, i_8] + T1[i_6, i_7]*t0[i_6, i_8]  {id=insn_5, priority=14}
│  end i_7, i_6, i_8
│  for i_9
└↱   t9[i_9] = 0.0  {id=insn_6, priority=13}
 │ end i_9
 │ for i35, i34
↱└     t5[i34, i35] = t3[i34, i35]  {id=insn_7, priority=12}
│  end i35, i34
└↱ SubArrayRef((i36, i37), (t5[i36, i37])) = inv(SubArrayRef((i34, i35), (t5[i34, i35])))        iority=11}
 │ for i42, i43
↱└     t7[i42, i43] = T1[i42, i43]  {id=insn_9, priority=10}
│  end i42, i43
└↱ SubArrayRef((i44, i45), (t7[i44, i45])) = inv(SubArrayRef((i42, i43), (t7[i42, i43])))        n_10, priority=9}
 │ for i_11, i_10
↱└     t9[i_11] = t9[i_11] + t4[i_10, i_11]*VecTemp1[i_10]  {id=insn_11, priority=8}
│  end i_11, i_10
│  for i_12, i_13
└↱     t10[i_13] = 0.0  {id=insn_12, priority=7}
 │   end i_13
 │   for i_14, i_15
↱└       t10[i_15] = t10[i_15] + t7[i_12, i_14]*T0[i_14, i_15]  {id=insn_13, priority=6}
│    end i_14, i_15
└↱   t12 = 0.0  {id=insn_14, priority=5}
↱└   t11 = 0.0  {id=insn_15, priority=4}
│    for i_16
└↱     t11 = t11 + t5[i_12, i_16]*VecTemp0[i_16]  {id=insn_16, priority=3}
 │   end i_16
 │   for i_17
↱└     t12 = t12 + t10[i_17]*VecTemp0[i_17]  {id=insn_17, priority=2}
│    end i_17
└    output[i_12] = output[i_12] + t12 + t9[i_12] + t11  {id=insn_18, priority=1}
   end i_12
---------------------------------------------------------------------------
```

Figure D.2: Annotated output from aggressive unary op nesting example, after optimisation

Appendix E

Appendix for Structured Sparse Local Tensors

E.1 Local kernel

```
KERNEL: form00_cell_integral_otherwise
---------------------------------------------------------------------------
ARGUMENTS:
A: type: np:dtype('float64'), shape: (6, 6), dim_tags: (N1:stride:6, N0:stride:1) aspace:
    global
coords: type: np:dtype('float64'), shape: (6), dim_tags: (N0:stride:1) aspace: global
---------------------------------------------------------------------------
DOMAINS:
{ [j0] : 0 <= j0 <= 2 }
{ [k0] : 0 <= k0 <= 2 }
{ [ip] : 0 <= ip <= 2 }
{ [j0_0] : 0 <= j0_0 <= 2 }
{ [k0_0] : 0 <= k0_0 <= 2 }
{ [j0_1] : 0 <= j0_1 <= 2 }
{ [k0_1] : 0 <= k0_1 <= 2 }
---------------------------------------------------------------------------
INAME IMPLEMENTATION TAGS:
ip: None
j0: None
j0_0: None
j0_1: None
k0: None
k0_0: None
k0_1: None
---------------------------------------------------------------------------
TEMPORARIES:
t0: type: np:dtype('float64'), shape: (3, 3), dim_tags: (N1:stride:3, N0:stride:1) scope:
    local
t1: type: np:dtype('float64'), shape: () scope:local
t2: type: np:dtype('float64'), shape: () scope:local
t3: type: np:dtype('float64'), shape: () scope:local
t4: type: np:dtype('float64'), shape: (3), dim_tags: (N0:stride:1) scope:local
t5: type: np:dtype('float64'), shape: () scope:local
t6: type: np:dtype('float64'), shape: () scope:local
t7: type: np:dtype('float64'), shape: (3, 3), dim_tags: (N1:stride:3, N0:stride:1) scope:
    local
---------------------------------------------------------------------------
INSTRUCTIONS:
t1 = (-1.0)*coords[0]  {id=insn, priority=9}
t2 = (-1.0)*coords[1]  {id=insn_0, priority=8}
```

```
t3 = abs((t1 + coords[2])*(t2 + coords[5]) + (-1.0)*(t1 + coords[4])*(t2 + coords[3]))  {
    id=insn_1, priority=7}
for k0, j0
    t7[j0, k0] = 0.0  {id=insn_2, priority=6}
end k0, j0
for ip
  t5 = t4[ip]*t3  {id=insn_3, priority=5}
  for j0_0
    t6 = t0[ip, j0_0]*t5  {id=insn_4, priority=4}
    for k0_0
      t7[j0_0, k0_0] = t7[j0_0, k0_0] + t0[ip, k0_0]*t6  {id=insn_5, priority=3}
end ip, j0_0, k0_0
for k0_1, j0_1
    A[0 + j0_1*2, 0 + k0_1*2] = A[j0_1*2, k0_1*2] + t7[j0_1, k0_1]  {id=insn_6, priority
    =2}
    A[1 + j0_1*2, 1 + k0_1*2] = A[1 + j0_1*2, 1 + k0_1*2] + t7[j0_1, k0_1]  {id=insn_7,
    priority=1}
end k0_1, j0_1
---------------------------------------------------------------------------
```

Listing E.1: Full original output for local kernel

```
KERNEL: form00_cell_integral_otherwise
---------------------------------------------------------------------------
ARGUMENTS:
A: type: np:dtype('float64'), shape: (2, 3, 3), dim_tags: (N2:stride:9, N1:stride:3, N0:
    stride:1) aspace: global
coords: type: np:dtype('float64'), shape: (6), dim_tags: (N0:stride:1) aspace: global
---------------------------------------------------------------------------
DOMAINS:
{ [j0] : 0 <= j0 <= 2 }
{ [k0] : 0 <= k0 <= 2 }
{ [ip] : 0 <= ip <= 2 }
{ [j0_0] : 0 <= j0_0 <= 2 }
{ [k0_0] : 0 <= k0_0 <= 2 }
{ [j0_1] : 0 <= j0_1 <= 2 }
{ [k0_1] : 0 <= k0_1 <= 2 }
{ [j1] : 0 <= j1 <= 1 }
---------------------------------------------------------------------------
INAME IMPLEMENTATION TAGS:
ip: None
j0: None
j0_0: None
j0_1: None
j1: None
k0: None
k0_0: None
k0_1: None
---------------------------------------------------------------------------
TEMPORARIES:
t0: type: np:dtype('float64'), shape: (3, 3), dim_tags: (N1:stride:3, N0:stride:1) scope:
    local
t1: type: np:dtype('float64'), shape: () scope:local
t2: type: np:dtype('float64'), shape: () scope:local
t3: type: np:dtype('float64'), shape: () scope:local
t4: type: np:dtype('float64'), shape: (3), dim_tags: (N0:stride:1) scope:local
t5: type: np:dtype('float64'), shape: () scope:local
t6: type: np:dtype('float64'), shape: () scope:local
t7: type: np:dtype('float64'), shape: (3, 3), dim_tags: (N1:stride:3, N0:stride:1) scope:
    local
---------------------------------------------------------------------------
INSTRUCTIONS:
t1 = (-1.0)*coords[0]  {id=insn, priority=8}
```

```
t2 = (-1.0)*coords[1]  {id=insn_0, priority=7}
t3 = abs((t1 + coords[2])*(t2 + coords[5]) + (-1.0)*(t1 + coords[4])*(t2 + coords[3]))  {
    id=insn_1, priority=6}
for k0, j0
    t7[j0, k0] = 0.0  {id=insn_2, priority=5}
end k0, j0
for ip
  t5 = t4[ip]*t3  {id=insn_3, priority=4}
  for j0_0
    t6 = t0[ip, j0_0]*t5  {id=insn_4, priority=3}
    for k0_0
      t7[j0_0, k0_0] = t7[j0_0, k0_0] + t0[ip, k0_0]*t6  {id=insn_5, priority=2}
end ip, j0_0, k0_0
for j0_1, j1, k0_1
      A[0 + j1, 0 + j0_1, 0 + k0_1] = A[j1, j0_1, k0_1] + t7[j0_1, k0_1]  {id=insn_6,
    priority=1}
end j0_1, j1, k0_1
---------------------------------------------------------------------------
```

Listing E.2: Full final output for local kernel

E.2 Code generated from PyOP2 wrapper

```
#include <math.h>
#include <petsc.h>
#define LOOPY_CALL_WITH_INTEGER_TYPES(MACRO_NAME) \
    MACRO_NAME(int8, char) \
    MACRO_NAME(int16, short) \
    MACRO_NAME(int32, int) \
    MACRO_NAME(int64, long)
#define LOOPY_DEFINE_FLOOR_DIV_POS_B(SUFFIX, TYPE) \
    inline TYPE loopy_floor_div_pos_b_##SUFFIX(TYPE a, TYPE b) \
    { \
        if (a<0) \
            a = a - (b-1); \
        return a/b; \
    }
LOOPY_CALL_WITH_INTEGER_TYPES(LOOPY_DEFINE_FLOOR_DIV_POS_B)
#undef LOOPY_DEFINE_FLOOR_DIV_POS_B
#undef LOOPY_CALL_WITH_INTEGER_TYPES
#include <stdint.h>
#include <stdint.h>

void wrap_form00_cell_integral_otherwise(int32_t const start, int32_t const end, Mat
  const mat0, double const *__restrict__ dat0, int32_t const *__restrict__ map0)
{
  double const form_t0[3 * 3] = { 0.6666666666666669, 0.16666666666666663,
  0.16666666666666666, 0.16666666666666674, 0.16666666666666663, 0.6666666666666665,
  0.16666666666666669, 0.6666666666666666, 0.16666666666666663 };
  double form_t1;
  double form_t2;
  double form_t3;
  double const form_t4[3] = { 0.16666666666666666, 0.16666666666666666,
  0.16666666666666666 };
  double form_t5;
  double form_t6;
  double form_t7[3 * 3];
  double t0[3 * 2];
  double t1[3 * 2 * 3 * 2];

  for (int32_t n = start; n <= -1 + end; ++n)
```

```c
    {
      for (int32_t i2 = 0; i2 <= 2; ++i2)
        for (int32_t i3 = 0; i3 <= 1; ++i3)
          for (int32_t i4 = 0; i4 <= 2; ++i4)
            for (int32_t i5 = 0; i5 <= 1; ++i5)
              t1[12 * i2 + 6 * i3 + 2 * i4 + i5] = 0.0;
      for (int32_t i0 = 0; i0 <= 2; ++i0)
        for (int32_t i1 = 0; i1 <= 1; ++i1)
          t0[2 * i0 + i1] = dat0[2 * map0[3 * n + i0] + i1];
      /* no-op (insn=form__start) */
      form_t1 = -1.0 * t0[0];
      form_t2 = -1.0 * t0[1];
      form_t3 = fabs((form_t1 + t0[2]) * (form_t2 + t0[5]) + -1.0 * (form_t1 + t0[4]) * (form_t2 + t0[3]));
      for (int32_t form_k0 = 0; form_k0 <= 2; ++form_k0)
        for (int32_t form_j0 = 0; form_j0 <= 2; ++form_j0)
          form_t7[3 * form_j0 + form_k0] = 0.0;
      for (int32_t form_ip = 0; form_ip <= 2; ++form_ip)
      {
        form_t5 = form_t4[form_ip] * form_t3;
        for (int32_t form_j0_0 = 0; form_j0_0 <= 2; ++form_j0_0)
        {
          form_t6 = form_t0[3 * form_ip + form_j0_0] * form_t5;
          for (int32_t form_k0_0 = 0; form_k0_0 <= 2; ++form_k0_0)
            form_t7[3 * form_j0_0 + form_k0_0] = form_t7[3 * form_j0_0 + form_k0_0] + form_t0[3 * form_ip + form_k0_0] * form_t6;
        }
      }
      for (int32_t form_k0_1 = 0; form_k0_1 <= 2; ++form_k0_1)
        for (int32_t form_j0_1 = 0; form_j0_1 <= 2; ++form_j0_1)
        {
          t1[12 * form_j0_1 + 2 * form_k0_1] = t1[12 * form_j0_1 + 2 * form_k0_1] + form_t7[3 * form_j0_1 + form_k0_1];
          t1[36 * ((7 + 12 * form_j0_1 + 2 * form_k0_1) / 36) + 12 * (1 + form_j0_1 + loopy_floor_div_pos_b_int32(-5 + 2 * form_k0_1, 12) + -3 * ((7 + 12 * form_j0_1 + 2 * form_k0_1) / 36)) + 6 * (-1 + (1 + 2 * form_k0_1) / 6 + -2 * loopy_floor_div_pos_b_int32(-5 + 2 * form_k0_1, 12)) + 2 * (form_k0_1 + -3 * ((1 + 2 * form_k0_1) / 6)) + 1] = t1[36 * ((7 + 12 * form_j0_1 + 2 * form_k0_1) / 36) + 12 * (1 + form_j0_1 + loopy_floor_div_pos_b_int32(-5 + 2 * form_k0_1, 12) + -3 * ((7 + 12 * form_j0_1 + 2 * form_k0_1) / 36)) + 6 * (-1 + (1 + 2 * form_k0_1) / 6 + -2 * loopy_floor_div_pos_b_int32(-5 + 2 * form_k0_1, 12)) + 2 * (form_k0_1 + -3 * ((1 + 2 * form_k0_1) / 6)) + 1] + form_t7[3 * form_j0_1 + form_k0_1];
        }
      /* no-op (insn=statement3) */
      MatSetValuesBlockedLocal(mat0, 3, &(map0[3 * n]), 3, &(map0[3 * n]), &(t1[0]), ADD_VALUES);
    }
  }
```

Listing E.3: Original code generated from PyOP2 wrapper.

```c
#include <math.h>
#include <petsc.h>
#include <stdint.h>
#include <stdint.h>

void wrap_form00_cell_integral_otherwise(int32_t const start, int32_t const end, Mat const mat0, double const *__restrict__ dat0, int32_t const *__restrict__ map0)
{
  double const form_t0[3 * 3] = { 0.6666666666666669, 0.16666666666666663, 0.16666666666666666, 0.16666666666666674, 0.16666666666666663, 0.6666666666666665, 0.16666666666666669, 0.6666666666666666, 0.16666666666666663 };
  double form_t1;
double form_t2;
double form_t3;
double const form_t4[3] = { 0.1666666666666666, 0.1666666666666666,
  0.1666666666666666 };
double form_t5;
double form_t6;
double form_t7[3 * 3];
double t0[3 * 2];
double t1[2 * 3 * 3];

for (int32_t n = start; n <= -1 + end; ++n)
{
  for (int32_t i2 = 0; i2 <= 2; ++i2)
    for (int32_t i3 = 0; i3 <= 1; ++i3)
      for (int32_t i4 = 0; i4 <= 2; ++i4)
        t1[6 * i2 + 3 * i3 + i4] = 0.0;
  for (int32_t i0 = 0; i0 <= 2; ++i0)
    for (int32_t i1 = 0; i1 <= 1; ++i1)
      t0[2 * i0 + i1] = dat0[2 * map0[3 * n + i0] + i1];
  /* no-op (insn=form__start) */
  form_t1 = -1.0 * t0[0];
  form_t2 = -1.0 * t0[1];
  form_t3 = fabs((form_t1 + t0[2]) * (form_t2 + t0[5]) + -1.0 * (form_t1 + t0[4]) * (
  form_t2 + t0[3]));
  for (int32_t form_k0 = 0; form_k0 <= 2; ++form_k0)
    for (int32_t form_j0 = 0; form_j0 <= 2; ++form_j0)
      form_t7[3 * form_j0 + form_k0] = 0.0;
  for (int32_t form_ip = 0; form_ip <= 2; ++form_ip)
  {
    form_t5 = form_t4[form_ip] * form_t3;
    for (int32_t form_j0_0 = 0; form_j0_0 <= 2; ++form_j0_0)
    {
      form_t6 = form_t0[3 * form_ip + form_j0_0] * form_t5;
      for (int32_t form_k0_0 = 0; form_k0_0 <= 2; ++form_k0_0)
        form_t7[3 * form_j0_0 + form_k0_0] = form_t7[3 * form_j0_0 + form_k0_0] +
  form_t0[3 * form_ip + form_k0_0] * form_t6;
    }
  }
  for (int32_t form_k0_1 = 0; form_k0_1 <= 2; ++form_k0_1)
    for (int32_t form_j1 = 0; form_j1 <= 1; ++form_j1)
      for (int32_t form_j0_1 = 0; form_j0_1 <= 2; ++form_j0_1)
        t1[3 * form_j0_1 + form_k0_1 + 9 * form_j1] = t1[3 * form_j0_1 + form_k0_1 + 9 *
   form_j1] + form_t7[3 * form_j0_1 + form_k0_1];
  /* no-op (insn=statement3) */

  const PetscInt x_indices[] = {2*map0[3 * n], 2*(map0[3 * n]+1), 2*(map0[3 * n]+2)};
  const PetscInt y_indices[] = {2*map0[3 * n]+1, 2*(map0[3 * n]+1)+1, 2*(map0[3 * n]+2)
  +1};

  MatSetValuesLocal(mat0, 3, &(x_indices), 3, &(x_indices), &(t1[0]), ADD_VALUES);
  MatSetValuesLocal(mat0, 3, &(y_indices), 3, &(y_indices), &(t1[9]), ADD_VALUES);
}
}
```

Listing E.4: Full desired code generated from PyOP2 wrapper.

Bibliography

[1] David Ham. Finite element course. https://www.youtube.com/playlist?list=PLh_UinHuMhzJlmqHU6LcFJ8YT9mRivz4J, 2019.

[2] J E Marsden, L Sirovich, S S Antman, G Iooss, P Holmes, D Barkley, M Dellnitz, and P Newton. *The Mathematical theory of finite element methods*, volume 46. 2003.

[3] Graham Robert Markall. Multilayered Abstractions for Partial Differential Equations. 2013.

[4] Thomas H. Gibson, Lawrence Mitchell, David A. Ham, and Colin J. Cotter. Slate: extending Firedrake's domain-specific abstraction to hybridized solvers for geoscience and beyond. pages 1–40, 2018.

[5] Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T.T. McRae, Gheorghe Teodor Bercea, Graham R. Markall, and Paul H.J. Kelly. Firedrake: Automating the finite element method by composing abstractions. *ACM Transactions on Mathematical Software*, 43(3), 2016.

[6] Eigen is a c++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms. http://eigen.tuxfamily.org/index.php?title=Main_Page.

[7] Andreas Klöckner. Loopy documentation. https://documen.tician.de/loopy/.

[8] Loopy github. https://github.com/inducer/loopy.

[9] Andreas Klockner. Loo.py: Transformation-based code generation for GPUs and CPUs. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 82–87, 2014.

[10] Francis P. Russell, Michael R. Mellor, Paul H.J. Kelly, and Olav Beckmann. An active linear algebra library using delayed evaluation and runtime code generation. *Science of Computer Programming*, 76(4):227–242, 2011.

[11] Miklos Homolya, Lawrence Mitchell, Fabio Luporini, and David A. Ham. TSFC: A structure-preserving form compiler. *SIAM Journal on Scientific Computing*, 40(3):C401–C428, 2018.

[12] Wikipedia. Finite element method. https://en.wikipedia.org/wiki/Finite_element_method, 2019.

[13] IEEE. The advantages of the finite element method. https://innovationatwork.ieee.org/the-advantages-of-fem/, 2019.

[14] Manor Tool. The benefits of finite element analysis in manufacturing. https://www.manortool.com/finite-element-analysis.

[15] Douglas H. Norrie. *A first course in the finite element method*, volume 3. 1987.

[16] Martin S. Alnæs, Anders Logg, Kristian B. Ølgaard, Marie E. Rognes, and Garth N. Wells. Unified form language: A domain-specific language for weak formulations of partial differential equations. *ACM Transactions on Mathematical Software*, 40(2), 2014.

[17] Ajay Harish. Finite element method - what is it? fem and fea explained. https://www.simscale.com/blog/2016/10/what-is-finite-element-method/, 2019.

[18] David Ham and Colin Cotter. Finite elements analysis and implementation.

[19] Galerkin method. https://en.wikipedia.org/wiki/Galerkin_method.

[20] Method of mean weighted residuals. https://en.wikipedia.org/wiki/Method_of_mean_weighted_residuals.

[21] Discontinuous galerkin method. https://en.wikipedia.org/wiki/Discontinuous_Galerkin_method.

[22] University of Cambridge. Nodes, elements, degrees of freedom and boundary conditions. https://www.doitpoms.ac.uk/tlplib/fem/node.php under https://creativecommons.org/licenses/by-nc-sa/2.0/uk/.

[23] University of Cambridge. Nodes, elements, degrees of freedom and boundary conditions. https://doitpoms.admin.cam.ac.uk/tlplib/fem/node.php?printable=1.

[24] Various authors. In finite element analysis, what exactly is the difference between degrees of freedom and nodes? https://www.quora.com/In-Finite-Element-Analysis-what-exactly-is-the-difference-between-degrees-of-freedom-and-nod

[25] Wikipedia. Lagrange multiplier. https://en.wikipedia.org/wiki/Lagrange_multiplier.

[26] Guyan reduction. https://en.wikipedia.org/wiki/Guyan_reduction.

[27] What is deal.ii? https://www.dealii.org/about.html.

[28] Febio software suite. https://febio.org/.

[29] A. Logg, K. A. Mardal, and G. Wells. *Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book (Lecture Notes in Computational Science and Engineering)*. 2012.

[30] Freefem: A high level multiphysics finite element software. https://freefem.org/.

[31] Getdp: A general environment for the treatment of discrete problems. http://getdp.info/.

[32] Kevin Long. Sundance 2.0 Tutorial @ Sandia National laboratories. (July), 2004.

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

[34] Miklós Homolya, Robert C. Kirby, and David A. Ham. Exposing and exploiting structure: optimal code generation for high-order finite element methods. 2017.

[35] Neil MacDonald, Elspeth Minty, Joel Malard, Tim Harding, Simon Brown, and Mario Antonioletti. Writing Message Passing Parallel Programs with MPI. (1.8.2).

[36] Sophia Vorderwuelbecke. Discussion about the latest slac developments. Personal communication.

[37] Faq: How does eigen compare to blas/lapack? http://eigen.tuxfamily.org/index.php?title=FAQ.

[38] Khaled Z. Ibrahim, Samuel W. Williams, Evgeny Epifanovsky, and Anna I. Krylov. Analysis and tuning of libtensor framework on multicore architectures. *2014 21st International Conference on High Performance Computing, HiPC 2014*, 2014.

[39] E. Solomonik, D. Matthews, J. Hammond, and J. Demmel. Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 813–824, 2013.

[40] Various authors. Tensorflow. https://www.tensorflow.org/.

[41] Markus Püschel, José M.F. Moura, Jeremy R. Johnson, David Padua, Manuela M. Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gačić, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–273, 2005.

[42] Charisee Chiw, Gordon L Kindlman, and John Reppy. EIN : An Intermediate Representation for Compiling Tensor Calculus. *Compilers for Parallel Computing*, 2016.

[43] Charisee Chiw, Gordon Kindlmann, John Reppy, Lamont Samuels, and Nick Seltzer. Diderot: A parallel DSL for image analysis and visualization. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 111–120, 2012.

[44] Dune numerics. https://www.dune-project.org/.

[45] Dominic Kempf, René Heß, Steffen Müthing, and Peter Bastian. Automatic Code Generation for High-Performance Discontinuous Galerkin Methods on Modern Architectures. 2018.

[46] Lawrence Mitchell and Eike Hermann Müller. High level implementation of geometric multigrid solvers for finite element problems: Applications in atmospheric modelling. *Journal of Computational Physics*, 327:1–18, 2016.

[47] Tianjiao Sun, Lawrence Mitchell, Kaushik Kulkarni, Klockner Andreas, David A. Ham, and Paul H.J. Kelly. A study of vectorization for matrix-free finite element methods. 2019.

[48] Lapack — linear algebra package. http://www.netlib.org/lapack/.

[49] Wikipedia: Basic linear algebra subprograms. https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms.

[50] Benchmark. http://eigen.tuxfamily.org/index.php?title=Benchmark.

[51] Paul S. Wang. FINGER: A Symbolic System for Automatic Generation of Numerical Programs in Finite Element Analysis. *Journal of Symbolic Computation*, 2(3):305–316, 1986.

[52] Francis R. Russell and Paul H.J Kelly. Optimized Code Generation for Finite Element Local Assembly Using Symbolic Manipulation. *ACM Transactions on Mathematical Software*, 39(4), 2013.

[53] John Ellson, Emden Gansner, Yifan Hu, Erwin Janssen, and Stephen North. Graphviz. https://graphviz.org/.

[54] Firedrake. Firedrake: Supported finite elements. https://www.firedrakeproject.org/variational-problems.html#supported-finite-elements.

[55] David Ham. Discussion about finite element families. Personal communication, June 2020.

[56] Firedrake github. https://github.com/firedrakeproject.

[57] Gheorghe-Teodor Bercea. Improving high performance computing using code generation and compilation techniques. page 254, 2017.

[58] pyop2 package. https://op2.github.io/PyOP2/pyop2.html.

[59] Facet (geometry). https://en.wikipedia.org/wiki/Facet_(geometry).

[60] Gheorghe Teodor Bercea, Andrew T.T. McRae, David A. Ham, Lawrence Mitchell, Florian Rathgeber, Luigi Nardi, Fabio Luporini, and Paul H.J. Kelly. A structure-exploiting numbering algorithm for finite elements on extruded meshes, and its performance evaluation in Firedrake. *Geoscientific Model Development*, 9(10):3803–3815, 2016.

[61] Fabio Luporini, Ana Lucia Varbanescu, Florian Rathgeber, Gheorghe Teodor Bercea, J. Ramanujam, David A. Ham, and Paul H.J. Kelly. Cross-loop optimization of arithmetic intensity for finite element local assembly. *ACM Transactions on Architecture and Code Optimization*, 11(4), 2014.

[62] Francis Russel. Exploring symbolic manipulation and other code generation techniques for finite element local assembly.

[63] Fabio Luporini, David A. Ham, and Paul H.J. Kelly. An algorithm for the optimization of finite element integration loops. *ACM Transactions on Mathematical Software*, 44(1):1–25, 2017.

[64] Rolf Stenberg. Postprocessing schemes for some mixed finite elements. *ESAIM: Mathematical Modelling and Numerical Analysis*, 25(1):151–167, 1991.

[65] Affine transformation.

[66] Wikipedia. Krylov subspace. https://en.wikipedia.org/wiki/Krylov_subspace.

[67] Brendan Gregg. Working set size estimation. http://www.brendangregg.com/wss.html.