IMPERIAL COLLEGE LONDON

DEPARTMENT OF MATHEMATICS

COMPOSABLE CODE GENERATION FOR HIGH ORDER, COMPATIBLE FINITE ELEMENT METHODS

Author:
Sophia Vorderwuelbecke

Supervisors:
Dr. David A. Ham, Prof. Colin C. Cotter

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy
in the Department of Mathematics and in the Fluid Dynamics CDT
at Imperial College London

January 16, 2023

Declaration of Authorship

I certify that this thesis, and the research to which it refers, are the product of my own work, and that any ideas or quotations from the work of other people, published or otherwise, are fully acknowledged in accordance with the standard referencing practices of the discipline.

January 16, 2023, Sophia Vorderwuelbecke

COMPOSABLE CODE GENERATION FOR HIGH ORDER, COMPATIBLE FINITE ELEMENT METHODS

It has been widely recognised in the HPC communities across the world, that exploiting modern computer architectures, including exascale machines, to a full extent requires software communities to adapt their algorithms. Computational methods with a high ratio of floating point operations to bandwidth are favorable. For solving partial differential equations, which can model many physical problems, high order finite element methods can calculate approximations with a high efficiency when a good solver is employed. Matrix-free algorithms solve the corresponding equations with a high arithmetic intensity. Vectorisation speeds up the operations by calculating one instruction on multiple data elements.

Another recent development for solving partial differential are compatible (mimetic) finite element methods. In particular with application to geophysical flows, compatible discretisations exhibit desired numerical properties required for accurate approximations. Among others, this has been recognised by the UK Met office and their new dynamical core for weather and climate forecasting is built on a compatible discretisation. Hybridisation has been proven to be an efficient solver for the corresponding equation systems, because it removes some inter-elemental coupling and localises expensive operations.

This thesis combines the recent advances on vectorised, matrix-free, high order finite element methods in the HPC community on the one hand and hybridised, compatible discretisations in the geophysical community on the other. In previous work, a code generation framework has been developed to support the localised linear algebra required for hybridisation. First, the framework is adapted to support vectorisation and further, extended so that the equations can be solved fully matrix-free. Promising performance results are completing the thesis.

Acknowledgements

Firstly and most importantly, I would like to thank Dr David A. Ham for his continuous support and guidance over the past years. David's technical excellence, organisational talent and strong interpersonal skills make him an invaluable group leader and supervisor. I would like to express equal gratitude for my second supervisor Prof Colin C. Cotter. Colin is collaborating and supervising with outstanding kindness and patience. I hold Colin in high regard for sharing his deep knowledge about numerical methods and computational fluid dynamics at various stages of my project. Even if I could revert any of my past decisions, choosing to work with David and Colin would certainly not be one of them.

Further, I feel indebted to all current and former members of the Firedrake project. Many thanks to Karina Kowalczyk, Lawrence Mitchell, Connor Ward, Jack Betteridge, Koki Sagiyama, Nacime Bouziani, Reuben Nixon-Hill and Paul Kelly for any technical and leisurely, off- and online conversations creating a team atmosphere which felt exceptional in an academic setting. My gratitude extends to my collaboration with Kaushik Kulkarni, one of Firedrake's backend developers. Kaushik's responses to my questions, our discussions about our software, and his quick fixes to the code generation were indispensable for my work, and I greatly enjoyed working together.

Although our time together was limited through the Covid19-pandemic, I cannot express how glad I am to count my fellow cohort members of the Fluid dynamics CDT as my friends. The intelligence and diversity in character in this cohort were stunning me daily. In particular, I would like to thank Geraldine Regnier for our deep friendship. Geraldine, the most intelligent person I have ever met, will stay my idol for the rest of my life. My thanks also go to Ali Arslan, James Hammond, Sam Hughes, Andy Killeen, Henrik Stumberg Larssen and Lorna Barron for stomaching our daily portions of 'Spaghetti Bernoulli' together. Further, I would like to express my appreciation of all members of the triathlon club at Imperial College, particularly Fiona Sander, Hazel May, Marion Artigaut and Edmund Jones. I appreciated every break away from College we had together.

Thanks to my family for giving me the unconditional freedom to chase my dreams without judgement. Thanks to my high-school friends Dorit, Moni, Nini and Katja for accepting me as the quirky one in their midst. Thanks to my spirit in another person, Laura Herz, for supporting me through a friendship that will last a lifetime.

Lastly, there is the one person who provided to this thesis what a concrete foundation is to a house. Thanks, Romain, for your understanding of all the stresses I felt during the PhD, thanks for your support in everyday life when I felt like there was time for nothing else besides my work, and most of all, thanks for keeping me steady when, frankly speaking, my work drove me nuts.

My funding was jointly provided by the Roth Scholarship of the Mathematics Department and the EPSRC Centre for Doctoral Training in Fluid Dynamics.

Copyright declaration

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Modelling climate change, generating energy from plasma fusion in a reactor, investigating how diseases spread in closed environments, controlling the distribution and production of electricity in smart grids, detecting exoplanets – all of these exemplary hot topics of the current era have one common factor. They are so complex, expensive or beyond physical reach, that mankind has to rely on computer calculations to find realistic approaches to solve the problems entailed in any of them. Nowadays, computer simulations are involved in virtually every sector of the economy and industry, having a big impact on everyone's life even if it often remains unnoticed. Applied natural and computer sciences are main drivers in the societal progress and this PhD thesis is a minuscule contribution in advancing the modern world.

A big part of computational simulations are numerical methods. Their need is driven by the fact that computers can only process a finite amount of discrete data. While the problems in the world seem to be continuous (can you imagine the ocean flowing in steps?), the best a computer can do is to give an approximation to the situation in the reality. Numerical methods are deployed to solve mathematical problems, which present models of the real-world problems, by discretising them and by finding solutions in a set of points.

A key characteristic of mathematics is that complicated concepts and relationships can be expressed by use of notations, similar to a language. This elegance of abstraction is especially vivid in the Finite Element Method (FEM), which is one of the most common, but also most complex numerical methods. FEM are frequently used for the spatial discretisation of partial differential equations (PDEs) modelling physical processes. The software framework Firedrake [1] exploits the FEM inherent abstractions and encapsulates mathematical and computational details in order to make the method accessible for other scientists.

FEM find application in numerous problems, for example in the area of fluid dynamics. The complexity of the underlying physical problems of PDEs translates into the numerical methods and requires a high amount of sophistication in order to achieve satisfying accuracy with an acceptable amount of resources. A concrete example for this is weather and climate prediction, where quantities have to be calculated on not only complex, but also massive domains and over various time scales. Influencing factors on the performance of these models can be of physical, mathematical or computational nature: examples are appropriate conservation of physical quantities, choice of spatial and temporal discretisation, and scalability (more resources yield a faster calculation). An implication of the many possible performance influencing factors is that improvements on the numerical model require different researchers' expertise. Thus, there is a need of abstractions for certain subareas. Separation of concerns, one of the core principles of Firedrake, eases improvements by experts in one field.

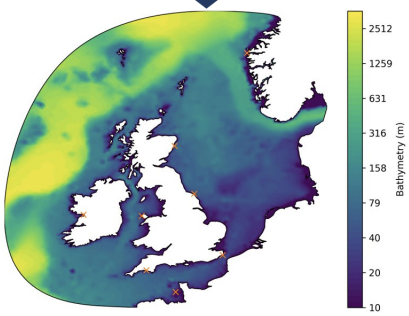
In this thesis, I contribute performance improvements for numerical methods, in particular for a special kind of FEM, a compatible and high order discretisation for PDEs. The optimisations of the computations are easily accessible for other scientists and applicable to many problems through automatic code generation with help of the software framework Firedrake [1].

$$\int_\Omega \nabla u \cdot \nabla v + uv \, \mathrm{d}x = \int_\Omega vf \, \mathrm{d}x$$

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

  for (int n = start; n <= -1 + end; ++n)
  {
    for (int i5 = 0; i5 <= 2; ++i5)
    {
      t4[i5] = dat2[map0[3 * n + i5]];
      for (int i6 = 0; i6 <= 1; ++i6)
        t3[2 * i5 + i6] = dat1[2 * map0[3 * n + i5] + i6];
    }
    for (int i1 = 0; i1 <= 2; ++i1)
      t2[i1] = 0.0;

    helmholtz(t2, t3, t4);
```



Bathymetry in the North Sea

Figure 1.1: Automatic code generation for FEM. The picture is extracted from `https://thetisproject.org/demos/demo_2d_north_sea.py.html`, accessed 26th September 2022.

1.2 Performance Improvements

Introducing performance improvements to Finite element methods is a complicated process and multiple factors have to considered. For the application scientists, who use FEM software to solve the PDEs modelling their physical problems, the ultimate goal is to achieve an approximation to the reality with a certain accuracy. The primary factor deciding what accuracy is sufficient for their problem is the needs of the scientific question. Those needs dictate the amount of resources, in particular how much time and computer power are required.

The accuracy of the approximation can be controlled with the choice of discretisation. In FEM, a decision must be made about how accuracy can be achieved. Both increasing the number of cells in the mesh or choosing higher order approximation polynomials can yield higher accuracy. Depending on that decision, the equation systems and, further, which algorithms are optimal to solve the set of equations efficiently changes. Whether a finer mesh or a higher order function space is the better mechanism to achieve higher accuracy depends on the type of computing power that is available to the application scientist. Lower order methods are sometimes used for their simplicity, however. Higher order methods either need enough regularity on the solutions or extra mechanisms like limiters. If the resulting equation systems are solved with iterative solvers, another contribution to the accuracy of the solutions is a specification of solver tolerances.

Suppose, the machine of the application scientist has a faster data transfer (higher bandwidth) than FLOP execution. Then, achieving the accuracy of the FEM on a refined mesh is the better choice. The reason for that is that the computation in the FEM is dominated by solving many small problems using a fine mesh, but lower order function spaces. In low order FEM, the ratio of the amount of computation to the amount of data which needs to be loaded, called the *arithmetic intensity* is low. The accuracy can be achieved fast if the data can be loaded fast.

In contrast, if the machine has lower bandwidth than FLOP execution, the accuracy of the approximation can be achieved faster with a discretisation using high order finite element function spaces. Then, the computations are dominated by solving fewer, but bigger problems. The arithmetic intensity is high. In order to achieve high accuracy fast, it becomes crucial to execute the operations on the data fast. The second argument for using high order FEM is a numerical one, namely their exponential convergence. Increasing the number of unknowns in the problem by choosing higher-order function spaces increases the accuracy of the solutions faster than by increasing the number of cells in the mesh. For further details refer to section 1.2.1.

Depending on the choice of discretisation, the structure of the resulting equation systems change, and different algorithms achieve higher performance. In particular, the choice between matrix-explicit and matrix-free methods depends on the choice of discretisation.

As previously mentioned, in low-order FEM, the operations are calculated on a small amount of data, the matrices are small. Matrix-explicit methods are sufficient. For higher order FEM, the matrices become larger, however, and the data movement should be minimised. Further, the cost of a sparse matrix-vector product scales with $\mathcal{O}(p^{2d})$, where p is approximation degree and d the dimension of the problem. Both data movement and the cost of matrix-vector product can be reduced by using matrix-free methods. In matrix-free methods, instead of loading and storing matrices, their product with a vector is assembled 'on the fly'. A fast execution of FLOPS is required. For further details refer to section 1.2.2.

If the wrong discretisation and algorithmic choices are made, only a fraction of the peak performance can be achieved. Ultimately, that means that the application scientist ends up either with an approximation less close to the reality, or with a longer wait time for the solutions. A longer wait time means a higher energy consumption. Therefore, the decision making in the code design process is paramount, particularly on supercomputing machines that consume a tremendous amount of energy. In order to leverage current computer architectures, which trend towards faster processors than data movements, methods exhibiting a high arithmetic intensity are favorable. Therefore, I introduce performance improvements only for high order Finite element methods. In this thesis, the introduced performance improvements emerge from creating code for new matrix-free solvers to the equations systems and from code optimisations for speeding up the calculations involved, refer to section 1.2.3.

The argument for using matrix-free, high order FEM being finished, we now focus on another aspect of the interplay between discretisation and performance. In this thesis, a special kind of discretisation called 'compatible' is considered. Compatible FEM provides a way to choose the discrete function spaces for quantities in coupled PDEs. Application scientists use compatible FEM because it ensures some numerical properties, e.g. conservation of physical quantities and stability, see section 1.2.1. The form of compatible FEM considered in this thesis ensures some continuity for one of the variables and therefore there is some coupling in the degrees of freedom. In parallel simulations the need for global reductions to calculate norms e.g. introduces a bottleneck. The processor communication bottleneck can be damped by using a so called hybridisation preconditioner [2] which removes some of the global coupling, for more details see section 1.2.2.

The coupling between architectures, discretisations and solving algorithms is one of the main motivations for code generation frameworks like Firedrake. Depending on the situation the users of Firedrake find themselves in, the code generation adapts, allowing for performance portability across application domains and computer architectures.

All components for highly performant, high order, compatible FEM are shown in figure 1.2.

Figure 1.2: Performance pyramid of high order, compatible FEM

1.2.1 Discretisation: Compatible, high order FEM

High accuracy in FEM can be achieved by choosing high order approximation polynomials under certain conditions. High order FEM are advantageous because they exhibit exponential convergence with the polynomial degree [3], and therefore come with a high efficacy. They are also characterised by a high arithmetic intensity which is favorable on modern computer architectures, as previously explained. The potential of high order discretisation has been recognised by many research groups providing computational solutions for FEM, such as Nek5000 [4], Nektar++ [5], DUNE [6] and deal. II [7], to name just a few.

Another factor influencing the choice of discretisation is the coupling of the variables in the PDE, for example velocity and pressure in the Navier Stokes equations. For coupled problems, the function space is chosen as a product space of the spaces for each quantity. Only the right combination of finite element spaces for the coupled quantities yields high-quality solutions, giving rise to a whole theory of 'compatible' function spaces, see [8], [9] [10], [11], [12], [13], [14] and more, based on Finite Element Exterior Calculus (FEEC) [15]. Compatible approximations turn out to be a sensible choice due to their excellent stability properties and elimination of spurious modes [11], as well as an exact conversation of physical quantities [11].

The benefits of compatible FEM have been acknowledged by the UK Met office, for example. The compatible discretisation from [16] is used in the new dynamical core GungHo [17] for weather and climate modelling. The software infrastructure allows for high order approximations and therefore, can benefit from the considerations in this thesis.

1.2.2 Solver: Fully matrix-free hybridisation

One possible application of fully matrix-free hybridisation is a mixed Poisson problem. Mixed Poisson problems are difficult to solve because, they have saddle-point structure and are indefinite. Schur complement solvers [18] can be used to eliminate one of the variables in the mixed system so that a system with a symmetric, positive-definite operator is solved instead.

Employing a Schur complement solver straightforwardly on a high order, mixed system would be problematic, however, since the system is large due to the high approximation degree. Further, mixed Poisson problems come with another difficulty, namely that the variables in the system are globally coupled. Some research has gone into approximate Schur complement preconditioners, e.g. in [19], but we focus on hybridisation. Hybridisation [20], [21] is a solving technique which reduces the amount of global coupling in a mixed system. This comes with the advantage that expensive operations like inverting a large, global matrix can be localised and thus, the operation is executed on a much smaller matrix. The prospect of performance benefits through hybridisation is also considered by the UK Met office [16].

In a high order FEM, even the local matrices are large and therefore, direct solves and inversions remain to be expensive operations even if hybridisation is used to localise them. Before my work, hybridised compatible FEM were only available for lower order approximations in Firedrake [2]. Its bottleneck is the store and load requirement of the operators and an explosion of the assembly cost. Matrix-free methods are ideally suited to resolve this issue. In matrix-free methods, parametrised operators are assembled many times instead of storing and reusing the original operator. Note that vectors not matrices are assembled in this case. In order to achieve fully matrix-free hybridisation, neither the global nor the local assembly of operators should be executed for matrices. Instead, the algorithms need to be transformed so that matrices only ever arise in the form of matrix-vector products. The products can be turned into the assembly of linear forms, and the only assembled objects are vector-valued. High order FEM are almost always solved with matrix-free methods as can be seen in the vast amount of literature on the support of high order DG in various FEM software packages and on developing suitable matrix-free solvers, e.g. in [22], [23] and [6] in DUNE, in [24] in Nektar++, in [25], [26] and [7] in deal. II and [27] in CEED. Using fully matrix-free hybridisation as a solver for compatible FEM is a new contribution.

In order to deliver high performance for fully matrix-free hybridisation, optimisations to speed up the computations, in particular sum-factorisation and vectorisation, had to be implemented for Slate in Firedrake beforehand, see section 1.2.3.

1.2.3 Code optimisations: Vectorisation and sum-factorisation

A performance-critical optimisation on modern architectures is vectorisation which involves calculations in the form of single instruction multiple data (SIMD). In vectorisation, loops over scalar operations are rewritten to vector operations, such that a single instruction can be executed parallelly on multiple data elements.

The global assembly in FEM presents significant opportunities for vectorisation, since the same local assembly function is applied to every mesh entity. Cross-element vectorisation, where not the innermost loop but the loop over the elements in the mesh is vectorised, has demonstrated consistently high performance for the assembly of various FE operator actions in [28] in Firedrake.

Vectorisation is particularly suited for high order FEM which exhibit a high arithmetic intensity and for matrix-free methods where only vectors are explicitly assembled. This has been recognised as part of most of the previous literature on matrix-free solvers and high order FEM. For frameworks like DUNE, automatic code generation for explicitly vectorised FEM is typically a complex change to the code generation process and so it is introduced separately to the support of high order and matrix-free methods, e.g. in [29].

In the same literature, it has been widely recognised that sum-factorisation in the local assembly is crucial for the performance of matrix-free, high order methods. Sum-factorisation lowers the complexity of the local assembly kernels by rewriting the loop nests and factoring out loop index independent calculations [30]. This is important for high order FEM where the loop bounds are large. If the equations systems of the high order discretisations are solved with matrix-free methods, the local assembly of actions (local matrix-free application) is called many times and therefore, a low complexity is favorable for high performance.

1.3 Contributions and dissemination

The main contribution of this thesis are performance improvements to compatible, high order FEM in the form of fully matrix-free, explicitly vectorised methods. The infrastructure supporting the performance improvements is implemented in Firedrake and involved the following changes to the code. The code is publicly available as open source software within Firedrake. The contributions are linked to the corresponding open-source pull requests in the appendix in section A.

1. *Two Stage Linear Algebra Compiler (TSSLAC)*
   The first contribution is an introduction of the new compiler TSSLAC, see section 3.2. The new compiler is required to ease the automatic code generation for the explicitly vectorised global assembly of Slate expressions. First, the compiler translates the local linear algebra language, called Slate, to an Einstein summation language, called GEM. Then, GEM is compiled to Loo.py, a language which eases the manipulation of loops, in the second stage. For more details see chapter 3.
2. *Slate Vectorisation*
   For the implementation of efficient vectorisation of Slate expressions the Loo.py package is used to batch the linear algebra kernels across elements, see section 3.3. The work is based on the vectorisation published in [28].
3. *Matrix-free methods in Slate*
   In order to make fully matrix-free methods available for users of the local linear algebra language, Slate, in Firedrake, multiple infrastructural changes had to be constituted.
   (a) Optimisations in the Slate compiler had to be introduced. A change of the assembly strategy of blocks, see section 4.2.1.3, and a reordering of the Slate expression, see section 4.2.1.2, were introduced in Firedrake.
   (b) Firedrake's domain specific languages (DSLs) had to be extended to support local preconditioners, see section 4.2.4.
   (c) Further extensions and translations of Firedrake's DSLs to support local actions, see section 4.2.2, and local matrix-free solvers, see section 4.2.3, have been implemented.
4. *Nesting of Schur complements in the Hybridisation preconditioner*
   The hybridisation preconditioner was extended to supported a nesting of Schur complements, see section 4.1.2.3.
5. *Local profiling infrastructure*
   In order to investigate the performance of the local linear algebra kernels, already existing logging infrastructure in Firedrake has been extended to measure timings of local kernels.
6. *Performance investigation framework*
   FE problems with hybridised high order compatible discretisations are expected to be efficiently solved with vectorised matrix-free methods. A framework to investigate the performance was introduced as part of chapter 5 and a first set of results is presented.

The complete package of preconditioned, matrix-free, vectorised, compatible, high order FEM should become a go-to method for a vast amount of engineering problems. The fact that these methods will be automatically available in Firedrake, embedded in DSLs, does not even leave the takes-too-long-to-program reason for why high order methods should not become widely used, also outside of academia.

The work presented in this thesis has already been communicated at a variety of workshops and conferences.

- 'Slate: Code Generation for Local Tensor Algebra Operations' at Dagstuhl Seminar 20111 – Tensor Computations: Applications and Optimization, March 2020
- 'An Introduction to Firedrake' at 2021 Code Performance Series: From analysis to insight, Performance analysis workshop 2021, January 202
- 'Generation of SIMD Vectorised Linear Algebra Operations on Finite Element Tensors' at SIAM CSE21, presenting a poster and a blitz video, March 2021
- 'Matrix-free, hybridised, compatible, high order finite element methods in Firedrake' at Firedrake '21, presenting a talk
- 'An automated framework for vectorised, matrix-free, hybridised, compatible, high order finite element methods' at SIAM PP 22, presenting a talk in the minisymposium 'Flexible, performance portable software for PDEs' (as main organiser)
- 'Matrix-free, hybridised, compatible, high order finite element methods in Firedrake' at ECCOMAS Congress 2021, presenting a talk in the minisymposium 'Advances in automatic code-generation software for simulations in Science and Engineering'

1.4 Thesis outline

In the background chapter 2 foundations are laid for the rest of the thesis. In particular, the theory of compatible, high order finite element methods and the resulting equation systems are presented in section 2.1. Further, hybridisation and Gopalakrishnan-Tan multigrid as solving and preconditioning techniques to these equation systems are explained in section 2.2. The code optimisations for high performance of high order FEM are described thereafter, in section 2.3, as well as the software framework Firedrake in which the optimisations are implemented, in section 2.4.

The following three chapters present technical contributions of the thesis. Chapter 3 presents the work on the support of vectorised linear algebra. Existing DSLs in Firedrake and their compilers are presented in section 3.1 as a precursor to the new compiler in section 3.2. The new compiler eased the path to vectorisation of Slate kernels, which is explained in section 3.3, and the results are presented in section 3.4. The focus in the next chapter is set on the support of matrix-free local linear algebra, the need for which is motivated in section 4.1. The new code infrastructure is explained in section 4.2, and its benefits over the previous infrastructure are proven in section 4.3. The penultimate chapter 5 provides a preliminary evaluation of the performance of a fully matrix-free, hybridised method for a mixed Poisson problem.

A summary and future work in chapter 6 conclude the thesis.

Chapter 2

Background

FEM provide a way to solve PDEs numerically. Starting from a PDE in strong form, various measures have to be taken to find a numerical solution. The strong form has to be converted into a variational form, see chapter 2.1.1 and a discretisation has to be chosen via a choice of mesh, see chapter 2.1.2.1, and function spaces, see chapter 2.1.2.2. The corresponding equation systems are derived by expressing the physical solutions in terms of basis functions, see chapter 2.1.3 and solvers have to be developed to calculate solutions for them, see chapter 2.2. The generated code for solving the equations can be optimised by expressing the math in a way which exploits current architectures more efficiently or by making use of lower-level performance optimisation, see chapter 2.3. The preliminaries for each stage of the FEM are presented in this chapter.

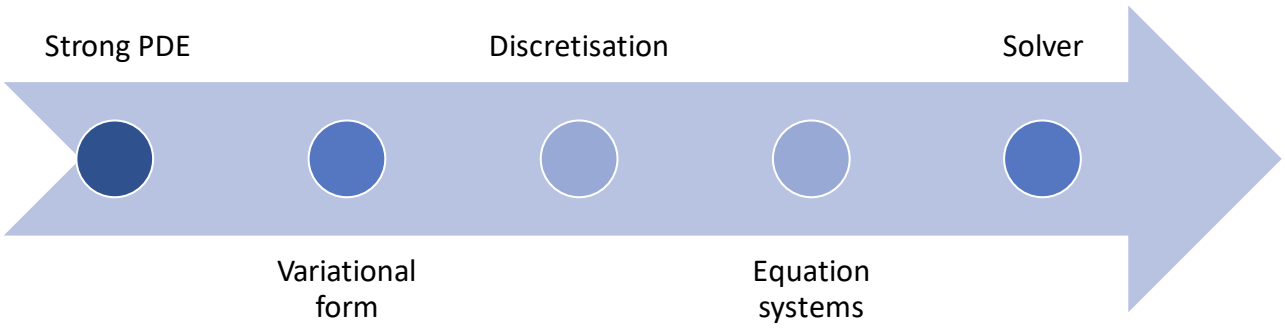


Figure 2.1: Flow chart of FEM

2.1 Discretisation of PDEs with Finite Element Methods

2.1.1 Variational form

A linear variational (or weak) form seeks a solution u that weakly satisfies the PDE for all test functions v, where $a(\cdot,\cdot)$ is a bilinear form and $l(\cdot)$ is a bounded linear functional.

$$\text{Find } u \in V \text{ such that } a(u,v) = l(v) \text{ for all } v \in V. \tag{2.1}$$

In this thesis, the functions u and v are elements of the same space, typical for Galerkin methods [31]. Given a strong form of a PDE depending on u, the weak form can be obtained by multiplication with the the test function v and integration by parts.

Ex. 2.1.1. *Poisson equation in variational form*
Consider the Poisson equation for some source function $f : \Omega \to V$ on a closed and bounded domain $\Omega \subset \mathbb{R}^d$ with the boundary condition on $\partial\Omega$ in strong form.

$$-\nabla^2 u = f \text{ on } \Omega \tag{2.2}$$

$$\nabla u \cdot n = 0 \text{ on } \partial\Omega \tag{2.3}$$

Its variational form is given by the following. The boundary integral disappears due to the choice of the boundary condition.

$$\text{Find } u \in V \text{ such that } \underbrace{\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}\mathbf{x}}_{a(u,v)} = \underbrace{\int_\Omega v f \, \mathrm{d}\mathbf{x}}_{l(v)} \qquad \forall v \in V \tag{2.4}$$

The function spaces used for V throughout the thesis in equation (2.1) are the following.

- The Sobolev space $W^{k,p}$ contains p-integrable functions with weak derivatives up to degree k which have a finite L^p-norm defined by $||x||_p = \left(\sum_{i\in I} |x_i|^p\right)^{1/p}$.
- $L^2(\Omega)$ is the set of square integrable functions on Ω:
$$L^2(\Omega) := \left\{ q : \Omega \to \mathbb{R} : \int_\Omega |q|^2 \, \mathrm{d}\mathbf{x} < \infty \right\}.$$
- $W^{k,2} = H^k(\Omega)$ is a special case of the Sobolev space and contains square-integrable functions with weak derivatives up to degree k which have a finite L^2-norm e.g.:
$$H^1(\Omega) := \left\{ q \in L^2(\Omega) : \nabla q \in \left[L^2(\Omega)\right]^d \right\}$$

- $H(\text{div})$ is the space of vector fields with square integrable divergence:

$$H(\text{div};\Omega) := \left\{ \boldsymbol{v} \in \left[L^2(\Omega)\right]^d : \nabla \cdot \boldsymbol{v} \in L^2(\Omega) \right\}$$

- $H(\text{curl})$ is the space of vector fields with square integrable curl:

$$H(\text{curl};\Omega) := \left\{ \boldsymbol{v} \in \left[L^2(\Omega)\right]^d : \nabla \times \boldsymbol{v} \in \left[L^2(\Omega)\right]^d \right\}$$

Another space worth mentioning is $\mathscr{P}_p(\Omega)$, the $(p+1)$-dimensional space of p-th degree polynomials on Ω, and its d-vector valued version denoted by $\left[\mathscr{P}_p(\Omega)\right]^d$.

2.1.2 Discretisation

2.1.2.1 Domain

In order to discretise the variational problem in equation (2.1), first the domain Ω has to be discretised by subdividing it into cells K. The discretisation of the domain is henceforth called a grid, mesh or tesselation. A mesh size parameter h has to be chosen as well as the geometry of the mesh. Let the tesselation be denoted by $\mathscr{T}_h := \bigcup_{K\in\Omega} K$ with $K_i \cap K_j = \delta K_{ij}$ or $\emptyset$.

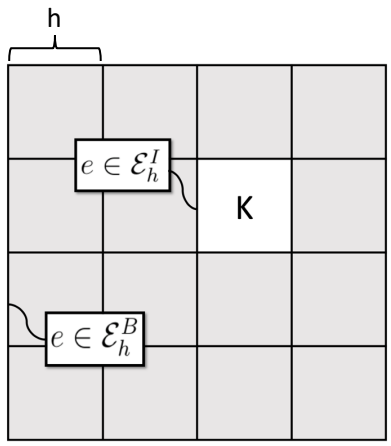


Figure 2.2 A quadrilateral, structured tesselation $\mathscr{T}_h$ of a 2D square domain Ω

Only quadrilateral and hexahedral meshes are considered for multiple reasons in this thesis. These meshes allow a simpler construction of the function spaces in section 2.1.2.2 by exploiting an inherent tensor product structure. Further down the line, exploiting the tensor product structure in the function space construction leads to expressions which can be sum-factorised to gain better performance, see section 2.3.2 for the technical details. Note that quadrilateral and hexahedral meshes are harder to generate for more geometrically complex geometries, however.

2.1.2.2 Function spaces

In a conforming finite element approximation the function space V is replaced by a discrete, finite dimensional function space $V_h \subset V$, whereas in nonconforming methods $V_h \not\subset V$ for example due to a relaxation of continuity in V_h. The discrete variational problem is given by the following.

$$\text{Find } u_h \in V_h \text{ such that } a(u_h, v_h) = l(v_h) \text{ for all } v_h \in V_h. \tag{2.5}$$

The discretisation of the global space V on the tesselation $\mathcal{T}$ is formulated per cell in the tesselation. The cellwise discretisation is a finite element $(K, V, \mathcal{N})$, introduced by Ciarlet in [32], see definition 2.1.1. The global space V_h is then $V_h := \{u \in V(\mathcal{T}_h) : u|_K \in V(K)\ \forall K \in \mathcal{T}_h\}$. There is a whole zoo of finite element spaces available coming with different convergence properties suitable for a wide range of problems.

Def. 2.1.1. *Ciarlet's finite element*

A finite element is a triple $(K, V, \mathcal{N})$ with:

- a bounded closed set $K \subset \mathbb{R}^d$ with piecewise smooth boundary ∂K;
- a finite dimensional (local) function space V on Kconsisting of real-valued functions with basis $\{\Psi_j\}_{j=0}^p$;
- a basis $\mathcal{N} = \{\Psi_i'\}_{i=0}^p = \{N_i\}_{i=0}^p$ for the dual space of V denoted by V', also called the set of degrees of freedom (DOFs)

With both local and global function spaces being defined, we focus on the elements in the function spaces, the functions themselves. The representation of the functions is important for the construction of the equation systems in (2.1.3) corresponding to the discretised problem. Every function in the local function space V in definition 2.1.1 can be represented by the basis functions in $\mathcal{N}$.

$$u(x) = \sum_{i=0}^{p} N_i(u)\Psi_i(\mathbf{x}) \qquad \text{for all } \mathbf{x} \in K, u \in V(K), K \in \mathcal{T}_h \tag{2.6}$$

The relation is more complicated for vector elements, see e.g. [33] and other evaluations than nodal evaluations where $N_i(\Psi_j) = \delta_{ij}$. Note that $V(K) = \text{span}(\{\Psi_i\}_{i=0}^p)$ for each K in the tesselation. Then, the global basis functions ψ are defined in terms of the local basis functions Ψ for each element in equation (2.7). In contrast, mapping from the local basis function to the global ones is more complicated and involves a coordinate change, which is encapsulated in a function which is called pullback.

$$\psi_{\hat{i}}(x)\big|_K := \Psi_i^K(x) \qquad \text{for all } \mathbf{x} \in \Omega, K \in \mathcal{T}_h \tag{2.7}$$

For details on the pullback, in particular for compatible FEM, refer to [21]. The global functions in the space V_h of dimension $n+1$ are defined with help of equation 2.7.

$$u_h(\mathbf{x}) = \sum_{\hat{i}=0}^{n} N_{\hat{i}}(u_{\hat{i}})\psi_{\hat{i}}(\mathbf{x}) \qquad \text{for all } \mathbf{x} \in \Omega, u_h \in V_h, K \in \mathcal{T}_h \tag{2.8}$$

Two finite elements used throughout the thesis are the continuous and the discontinuous Lagrange element on a cube, Q_p and DQ_p. How to construct the local spaces for those exploiting the tensor product structure of the cell, as previously mentioned in section 2.1.2.1, is explained in the example 2.1.2. For an description of the construction of various elements for different geometries and nodes refer to [21].

Ex. 2.1.2. *Continuous and discontinuous Lagrange elements on a unit interval, square and cube*

- Let $\mathcal{T}_h$ be a tessellation of a domain Ω with arbitrary dimension, where $K \in \mathcal{T}_h$ denotes a cell. In this example the cell is a unit cell $K = [0,1]^d$ with $d = 1,2,3$.
- Further, let $V(K)$ be a finite dimensional space built by p-th degree Lagrange polynomials. Since K is defined as a tensor product, the space can be defined as a tensor product of polynomial spaces on the interval $V(K) := \bigotimes_{i=0}^{d-1} \mathscr{P}_p([0,1])$.
- In this example, the set of nodes consists of point evaluations in the form of $N_i(u) := u(x_i)$ for $i \in \{0,\dots,p\}$ and $u \in V$ on a set of points $S := \{x_i\}_{i=0}^{p}$ of a cell. Since the cell in this example has tensor product structure, the set of nodes can be constructed through a tensor product of function evaluations. Let $x_i \in [0,1]$, and let N_i be the point evaluations on x_i, then the set of nodes is $\mathcal{N} = \left\{\bigotimes_{j=0}^{d-1} N_i\right\}$. Note that j is a counter for the dimensions. The set S can be constructed in various ways, refer to the chapter on quadrature points (and rules) (2.3.1) in [21] or 5.4 in [34].

If the points in S which lie on the boundary of K are shared between cells, the element $(K, V, \mathcal{N})$ is called a continuous Lagrange element, if not a discontinuous one. Continuous Lagrange elements are H^1-conforming, while discontinous Lagrange elements are L^2-conforming. When discontinuous Lagrange elements are used, the contribution on facets of neighbouring cell might differ from each other so that there is no global continuity enforced on the global space V_h.

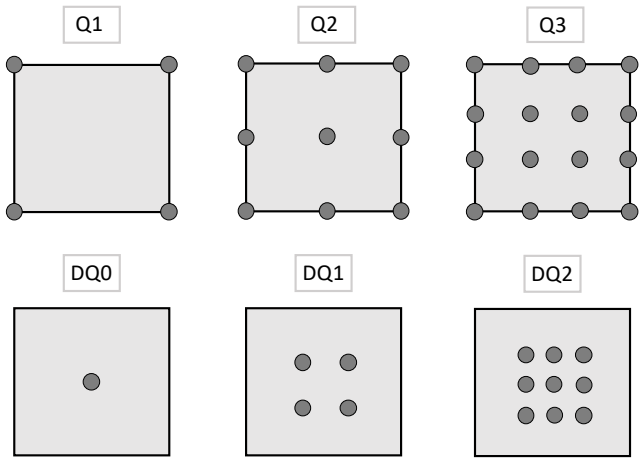


Figure 2.3: Lowest to next lowest order (left to right), continuous (top) and discontinuous (bottom) Lagrange elements on a quadrilateral cell. The round point represents a point evaluation. Note the indexing in: DQ_0 is the lowest order discontinuous, and Q_1 is the lowest order continuous element.

2.1.2.2.1 Compatible FEM For coupled equations, for example a velocity-pressure system like the Navier Stokes equations, mixed FEM are widely employed. In mixed FEM different FE spaces are used for the different variables in the system, which serves the purpose to account for different properties of each quantity. In compatible FEM, which are a subclass of mixed FEM, the different spaces are chosen in way that coupling them together results in a discretisation, which fulfills some wished for numerical properties, in particular stability and convergence [15], [35]. Compatible FEM are also known to to preserve geometric and topological structures [36]. The motivation for a discretisation with these properties is given in section 1.2.1. How to choose the space for each quantity, so that that e.g. stability is ensured, has been developed in the theory of finite element exterior calculus [15]. The key property that the spaces need to fulfil in order to be compatible is that they need to respect the L^2-DeRham complex.

Def. 2.1.2. *L^2-DeRham complex*
Let $\Omega \subset \mathbb{R}^d$ be a bounded domain, $V^k : \Omega \to \mathbb{R}^d$ be Hilbert spaces and d^k mappings which fulfill $d^k \circ d^{k-1} = 0$, then the L^2-DeRham complex is defined as follows.

$$0 \to V^0(\Omega) \xrightarrow{d^0} V^1(\Omega) \xrightarrow{d^1} \dots \xrightarrow{d^{n-1}} V^n(\Omega) \to 0 \tag{2.9}$$

Here the focus is set on compatible discretisations in 3D, but a more extensive explanation on compatible discretisations can be found in [21] or in [37].

Ex. 2.1.3. The only L^2 complex in $\mathbb{R}^3$ is the following [15].

$$0 \to H^1(\Omega) \xrightarrow{\nabla} H(\text{curl};\Omega) \xrightarrow{\nabla\times} H(\text{div};\Omega) \xrightarrow{\nabla\cdot} L^2(\Omega) \to 0 \tag{2.10}$$

With help of the L^2 -DeRham complex, which defines a mapping between the spaces of the continous problem, a definition of a compatible discretisation can be given.

Def. 2.1.3. *Compatible discretisation*
Let $V_h^k \subset V^k$ be discretisations and π^k be bounded projections for all k. Choosing a compatible discretisation means that the following diagram commutes with the chosen spaces V_h^k and the corresponding projections from the continuous spaces π_k.

$$V^0(\Omega) \xrightarrow{d^0} V^1(\Omega) \xrightarrow{d^1} \dots \xrightarrow{d^{n-1}} V^n(\Omega) \tag{2.11}$$

$$\Big\downarrow \pi^0 \qquad \Big\downarrow \pi^1 \qquad\qquad \Big\downarrow \pi^n \tag{2.12}$$

$$V_h^0(\Omega) \xrightarrow{d^0} V_h^1(\Omega) \xrightarrow{d^1} \dots \xrightarrow{d^{n-1}} V_h^n(\Omega) \tag{2.13}$$

Based on the definition in 2.1.3 that the diagram commutes, conditions for the functions in the discretised spaces can be specified, see for example the conditions for a 3D domain in example 2.1.4.

Ex. 2.1.4. *Compatibility in 3D*

Let Ω, $\mathcal{T}_h$ and V_h^k be defined as in previous sections. Further, let $V_h^0 \subset H^1$, $V_h^1 \subset H(\text{curl})$, $V_h^2 \subset H(\text{div})$ and $V_h^3 \subset L^2$. A discretisation based on the finite elements spaces V_h^k for $k = 0, \ldots, 3$, which is respecting the L^2-DeRham complex in three dimensions and is thus called compatible, is required to satisfy the following.

- $\nabla p \in V_h^1 \qquad \forall p \in V_h^0$
- $\nabla \times \mathbf{w} \in V_h^2 \qquad \forall \mathbf{w} \in V_h^1$
- $\nabla \cdot \mathbf{u} \in V_h^3 \qquad \forall \mathbf{u} \in V_h^2$
- The diagram in definition 2.1.3 commutes for the DeRham complex in example 2.1.3

The most famous examples for compatible FEM, including continuous and discontinuous Lagrange, Nedelec [38], Raviart-Thomas [39] and Brezzi-Douglas-Marini [40] on triangular and quadrilateral elements, are listed in [41]. A compatible choice of finite elements spaces on hexahedral cells, which respect the L^2 complex is $0 \to Q_q \xrightarrow{\nabla} NC_q^e \xrightarrow{\nabla\times} NC_q^f \xrightarrow{\nabla\cdot} DQ_{q-1} \to 0$. The order of the element is denoted by q. Note that NC is the finite element space in 3D developed by Nedelec in [38] as an extension of a Raviart Thomas element in 2D. NC^e are $H(\text{div})$ elements and NC^f are $H(\text{curl})$ elements. The compatible FEM which is based on this complex and is used throughout the thesis is henceforth called the Raviart-Thomas-Discontinuous-Galerkin (RT-DQ).

Ex. 2.1.5. *A compatible discretisation for example 2.1.1: RT-DQ in 3D*

Let $\mathcal{T}_h$ be a tessellation of a domain Ω in 3D, where $K \in \mathcal{T}_h$ denotes a cell in $\mathcal{T}_h$.

Let the local function space $V(K)$ be the smallest subset of $[P_{p+1}(K)]^3$ for which the divergence of the functions in V maps to a function in $P_p(K)$ and $Q(K) := P_p(K)$. Refer to [21] for a detailed construction of V for different cells geometries and dimensions.

The nodes in $\mathcal{N}$ are defined as follows. For all cells let V^e be the p-varying polynomials on the facets of the cell, and V^i be the $p-1$-varying vector valued polynomials on the interior of the cells. For a mathematical construction of V^e and V^i refer to [21].

$$\boldsymbol{v} \in V(K) : \mathcal{N}(\boldsymbol{v}) := \begin{cases} \int_e \boldsymbol{v} \cdot \boldsymbol{n} w \, \mathrm{d}s & \forall w \in V^e(e), \forall e \in \partial K \\ \int_K \boldsymbol{v} \cdot \boldsymbol{\psi} \, \mathrm{d}x & \forall \boldsymbol{\psi} \in V^i(i), \forall i \in K / \partial K, p+1 > 2 \end{cases} \tag{2.14}$$

The Raviart Thomas element has continuous normal in order to fulfill $H(\text{div};\Omega) \xrightarrow{\nabla\cdot} L^2(\Omega)$ as required by the DeRham complex in example 2.1.3, and discontinuous tangential components whereas the discontinuous Lagrange element is fully discontinuous.

The global, discretised function space is a product space $\Pi_h := V_h \times Q_h \subset H(\text{div}) \times L^2$ and consists of the space of $H(\text{div})$-conforming vector polynomials V_h by Raviart-Thomas and the space of the discontinous polynomials Q_h.

$$V_h := \{\boldsymbol{v} \in H(\text{div};\Omega) : \boldsymbol{v}|_K \in V(K) \ \forall K \in \mathcal{T}_h\} \tag{2.15}$$

$$Q_h := \{q \in L^2(\Omega) : q|_K \in Q(K) \ \forall K \in \mathcal{T}_h\} \tag{2.16}$$

A figure of this example in a lower dimension for an easier perception is presented in figure 2.4. A visualisation for higher dimension can be found for example in [41].

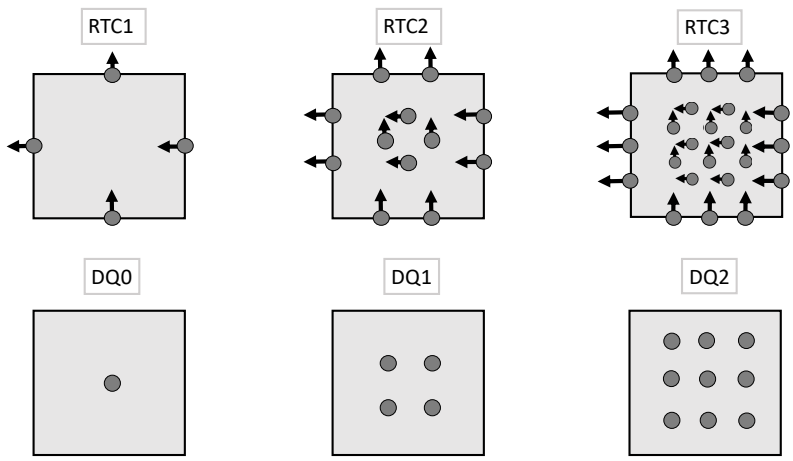


Figure 2.4: The Raviart-Thomas discontinous Lagrange FE pair on quadrilateral elements. Note that we follow the convention that RTC1 is the lowest order Raviart-Thomas element. [1]

[1] The picture is extracted from thesis submission 'A Higher Order Mixed Discontinuous Finite Element Method For Incompressible Flows' for fulfillment of the MRes Fluid Dynamics, by Sophia Vorderwuelbecke, submitted September 2019, Imperial College London.

2.1.3 Equation systems in FEM

The equation systems derived from the discretisation of a PDE with FE spaces are built considering that the FE functions can be represented in terms of their basis functions. Consider equation (2.8), and further let $\mathbf{A} = \left(a_{ij}\right)_{i,j\in S}$ and $\mathbf{b} = (b_i)_{i\in S}$ and $\tilde{\mathbf{u}} = \left(u_j\right)_{j\in S}$ for $S = \{0,\dots,n-1\}$, then the discretised variational problem from equation (2.5) can be rewritten into a linear equation system.

$$a(u_h, v_h) = l(v_h) \tag{2.17}$$

$$\Leftrightarrow \quad \sum_{i=0}^{n-1} \underbrace{a\left(\psi_j, \psi_i\right)}_{a_{ij}} u_j = \underbrace{l\left(\psi_i\right)}_{b_i} \tag{2.18}$$

$$\Leftrightarrow \quad \mathbf{A}\tilde{\mathbf{u}} = \mathbf{b} \tag{2.19}$$

Typically, the operators $\mathbf{A}$ and $\mathbf{b}$ in the equation system are calculated per cell K in the tesselation $\mathcal{T}_h$. The local contributions are distributed into the global matrix, thereafter. The collecting of the local contributions in the global operator is called the *global assembly*, see figure 2.5.

Based on equation (2.7), the local system matrix for each $K \in \mathcal{T}_h$ is defined by $A^K_{ij} := a_K\left(\Psi^K_i, \Psi^K_j\right)$, where a_K is the variational form on K, for all i and j. Building $\mathbf{A}^K$ is called the *local assembly*. The local assembly is mostly treated as a black-box in this thesis. Instead, most of the work focuses on improving the local linear algebra on the operators provided by the local assembly. Details on the local linear algebra operations can be found in the introduction to chapter 3.

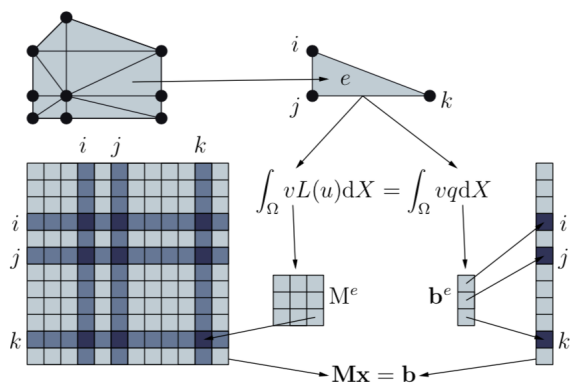


Figure 2.5: Global assembly with PyOP2 [1]. Local linear algebra operations can be executed on the local operators M^e and b^e before the global assembly.

2.2 Solver and Preconditioners

In the previous sections, the process to turn a physical problem into an equation system has been presented. Now, the aim is to provide background on the algorithms which are used to obtain solutions from those equation systems. Light is shed on the solvers and preconditioners for both local and global equation systems. The focus is set on methods which are expected to perform well for the problems emerging from compatible, high order FEM.

2.2.1 Preconditioned Krylov solvers

The equations system arising from FE discretisation can be written in a residual form as $F(u, v) = 0$ with $F(u, v) := A(u, v) - b(v)$, and are potentially nonlinear in u but not in this thesis. The linear systems can be solved for example with Krylov Subspace Methods (KSPs). Examples of KSPs are Generalized Minimal Residual (GMRES)[42] and Conjugate Gradient (CG)[43] methods. In KSPs a solution is found by minimising the residual over an r-th order Krylov subspace $\mathcal{K}_r$.

$$\mathcal{K}_r(A, b) := \text{span}\{b, Ab, \ldots, A^{r-1}b\}$$

Krylov subspace methods are almost always only efficient solvers under the provision of a suitable preconditioner since operators from PDE discretisations can have a large condition number. Instead of solving a system $F(u, v) = 0$ as mentioned above, the solution of an equivalent system $P^{-1}F(u, v) = 0$ is sought (with left preconditioning). In preconditioned Krylov subspace methods a solution to the equivalent system is found by calculating a residual first with $r_k = b - Ax_k$. In a next step, the preconditioner calculates $Py_k = r_k$ with the residual as a right-hand side, and then the solution is updated based on that. This is repeated until convergence. More generally, a preconditioner takes one linear system and solves another one with better numerical properties. It should be stressed that P must be easy to invert, yet approximate A well.

2.2.2 Solver of interest

For reasons covered in chapter 4.1, the equations of interest are based on a mixed Poisson problem. The system is solved most effectively with multiple solvers and preconditioners nested into each other, the best solver nesting is visualised in figure 2.6. The preconditioning technique known to perform well for the problem of interest is hybridisation (section 2.2.3). In the hybridisation preconditioner, a global system defined on the facets of the mesh has to be solved. The conjugate gradient method, refer to [44], preconditioned by Gopalakrishnan-Tan multigrid (section 2.2.4) is known for being a good algorithm for that, according to [45]. There are two more solvers nested into the GTMG, a smoother and a fine level solver. For the former the Chebyshev algorithm, refer to [44], is employed, whereas for the latter a direct method or geometric multigrid (section 2.2.4) can be used.

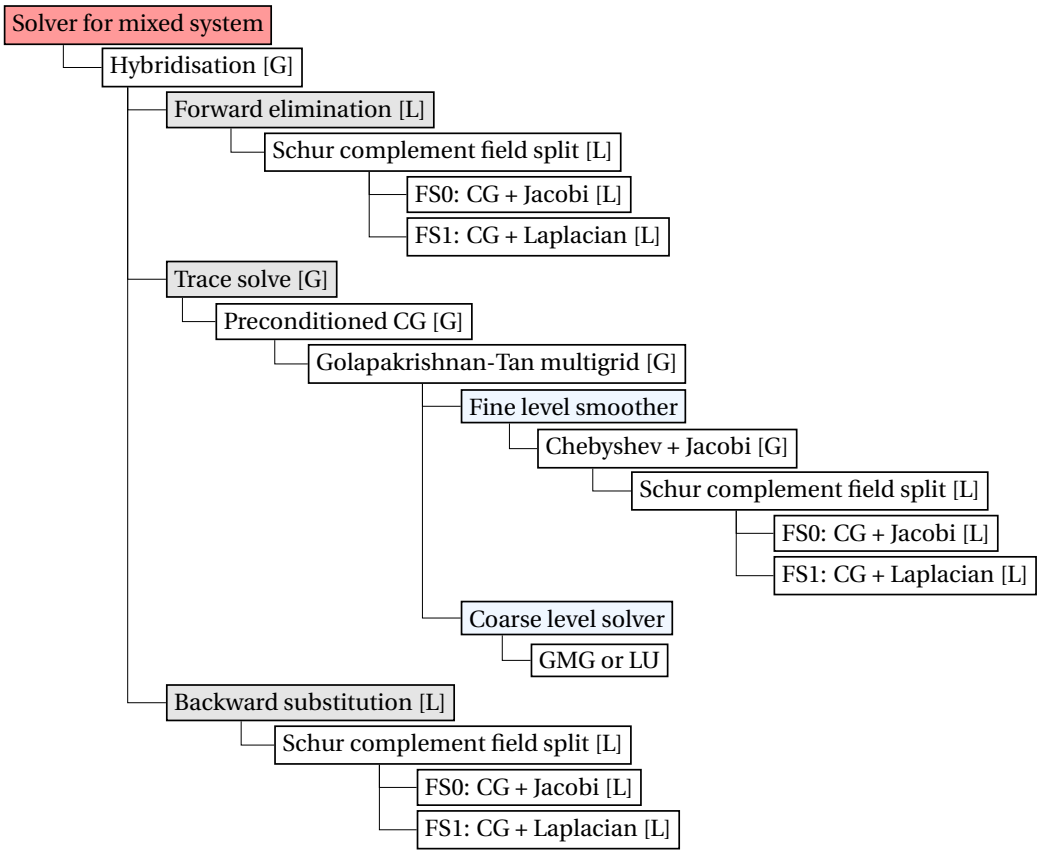


Figure 2.6: The expected to be best-performing nest of solvers in this thesis is represented in a tree. 'G' in square brackets denotes a global solve and 'L' a local solve (the difference between local and global solve is explained in section 2.2.3). Coloured boxes are components of the parent solver.

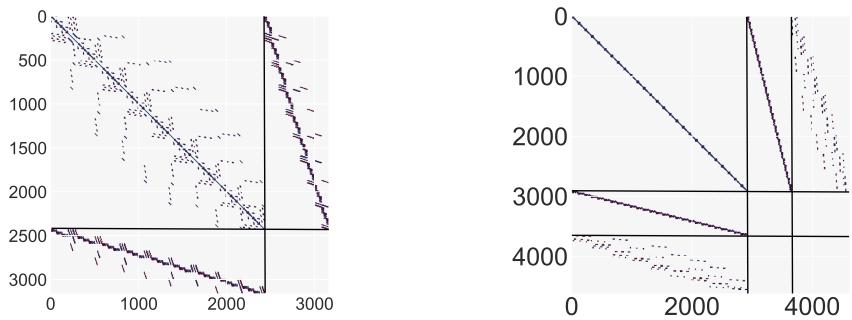
The hybridisation preconditioner is represented through linear algebra operations on the local matrices and involves inverse operations. If a fully matrix-free solving technique (see section 2.3.1 for an introduction) is applied, the local inverses in the hybridisation preconditioner are approximated. In this thesis, the approximation is calculated with a Conjugate gradient method preconditioned either by Jacobi or a Laplacian (for details refer to chapter 4.1).

2.2.3 Hybridisation

Hybridisation has been proven to be an efficient method to solve systems arising from compatible discretisations [21]. Hybridisation was first deployed for mixed formulations of second order elliptic PDEs [46], but it can also be used for discontinuous Galerkin methods [47]. [48] gives conditions for a system to be hybridisable. The main idea of hybridisation is to remove the inter-elemental coupling in the system matrix by moving the information about the coupling into a separate variable, a Lagrange multiplier. Then, the system matrix becomes block sparse and a local elimination can be applied. The elimination involves building Schur complements on the blocks of the hybridised system matrix, and the term 'local' in this context signifies that all linear algebra operations involved act on matrices which contain elementwise contributions to the finite element problem.

2.2.3.1 Breaking the spaces

The block sparse structure, see for example figure 2.7, is created by breaking the continuity in the mixed spaces. In figure 2.7a, a mixed system for two variables is displayed where the first variable U is discretised with a partially continuous space. In the matrix, the partial continuity becomes apparent by the contributions off the diagonal. After choosing the first variable in a space where the partial continuity is broken instead, the off-diagonal contributions are not visible anymore, compare for example the A_{00} block figure 2.7a and 2.7b. In the hybridised problem, the continuity of the original problem is still enforced but through a new variable Λ defined on the traces which acts as Lagrange multiplier.



(a) $\begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \begin{pmatrix} U \\ P \end{pmatrix} = \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$ **(b)** $\begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} U \\ P \\ \Lambda \end{pmatrix} = \begin{pmatrix} F_0 \\ F_1 \\ F_2 \end{pmatrix}$

Figure 2.7: Comparison of the mixed system, equation (2.26) (left), to the block sparse hybridised system, equation (2.29)-(2.31) (right), of the Poisson equation in example 2.2.1 on a $3 \times 3 \times 3$, hexahedral mesh discretised with $RTCF3 - DQ2$

An example of how to break a Raviart-Thomas space of order 2 defined on a two-cell quadrilateral mesh is visualised in figure 2.8 and the discontinuous space defined on the facets of the mesh for the Lagrange multiplier is also shown. A complete example of how to transform a PDE in strong form to a hybridised set of equations is shown in 2.2.1.

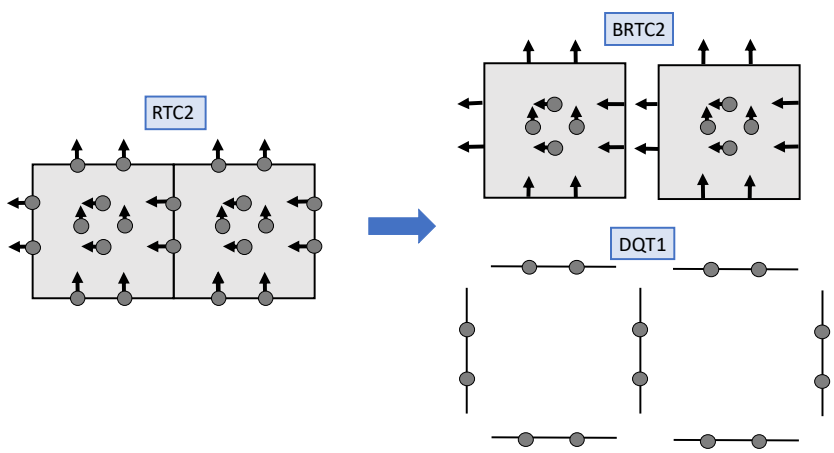


Figure 2.8: An example for breaking a partially continuous space (RTC2) into a fully discontinuous one (BRTC2), and the trace space (DQT1)

Ex. 2.2.1. *Hybridisation of a mixed Poisson equation*

1. Consider a mixed Poisson problem on a unit square $\Omega = [0,1] \times [0,1]$ with boundary $\partial\Omega = \partial\Omega_D \cup \partial\Omega_N$. Let $\mathbf{u} \in U$ and $p \in V$.

$$\mathbf{u} - \nabla p = 0 \qquad \text{on } \Omega \tag{2.20}$$
$$\nabla \cdot \mathbf{u} = -f \qquad \text{on } \Omega \tag{2.21}$$
$$p = p_0 \qquad \text{on } \partial\Omega_D \tag{2.22}$$
$$\mathbf{u} \cdot \mathbf{n} = g \qquad \text{on } \partial\Omega_N \tag{2.23}$$

   In this example consider the initial values as $p_0 = 0$ and $g = 0$.

2. The choice of discretisation is given by a compatible $U_h \times V_h \subset H(\text{div}) \times L^2$ space with a tesselation Ω_h and cells K.

$$U_h := \{\mathbf{w} \in H(\text{div};\Omega_h) : \mathbf{w}|_K \in U(K), \forall K \in \Omega_h, \mathbf{w} \cdot \mathbf{n} = \mathbf{0} \text{ on } \partial\Omega_h\} \tag{2.24}$$
$$V_h := \{\phi \in L^2(\Omega_h) : \phi|_K \in V(K), \forall K \in \Omega_h\} \tag{2.25}$$

   The discretised weak mixed form of the Poisson equation is defined in terms of the L^2 inner product $(\cdot,\cdot)$ considering the previously defined function spaces.

$$(\mathbf{u}_h, \mathbf{w})_{\Omega_h} + (\nabla \cdot \mathbf{w}, p_h)_{\Omega_h} + (\nabla \cdot \mathbf{u}_h, \phi)_{\Omega_h} = -(f, \phi)_{\Omega_h} \quad \forall\ (\mathbf{w}, \phi) \in U_h \times V_h \tag{2.26}$$

3. In order to transform the mixed form into a hybridisable form the continuity in the space for u has to be broken into intra and inter-elemental contributions $\hat{U}_h$ and U_h^{tr}. Note that $\mathscr{E}_h$ denotes the domain solely consisting of facets.

$$\hat{U}_h := \{\mathbf{w} \in L^2(\Omega_h) : \mathbf{w}|_K \in U(K), \forall K \in \Omega_h, \mathbf{w} \cdot \mathbf{n} = \mathbf{0} \text{ on } \partial\Omega_h\} \tag{2.27}$$
$$U_h^{tr} := \{\gamma \in L^2(\mathscr{E}_h) : \gamma|_e \in \mathscr{P}_k(e), \mathbf{w} \cdot \mathbf{n}|_{\mathbf{e}} = \mathbf{0}\ , \forall e \in \mathscr{E}_h, \forall \mathbf{w} \in U_h\} \tag{2.28}$$

   The hybridisable mixed Poisson equation is then given by the following. Note that $[[\cdot]]$ denotes the jump across the normal component of vector field on a facet.

$$(\mathbf{w}, \mathbf{u}_h)_{\Omega_h} + (\nabla \cdot \mathbf{w}, p_h)_{\Omega_h} + ([[\mathbf{w}]], \lambda_h)_{\partial\Omega_h \setminus \partial\Omega_D} = 0 \qquad \forall \mathbf{w} \in \hat{U}_h \tag{2.29}$$
$$(\phi, \nabla \cdot \mathbf{u}_h)_{\Omega_h} = -(\phi, f)_{\Omega_h}\ \forall \phi \in V_h \tag{2.30}$$
$$([[\mathbf{u}_h]], \gamma)_{\partial\Omega_h \setminus \partial\Omega_D} = 0 \qquad \forall \gamma \in U_h^{tr} \tag{2.31}$$

   The new equation system is visualised in figure 2.7b.

2.2.3.2 Static condensation

Static condensation is a solving technique where one or more variables of an equation system are eliminated. The elimination can be expressed in terms of a Schur complement factorisation. Consider an equation system in the following form.

$$\begin{pmatrix} A_{ee} & A_{ec} \\ A_{ce} & A_{cc} \end{pmatrix} \begin{pmatrix} X_e \\ X_c \end{pmatrix} = \begin{pmatrix} F_e \\ F_c \end{pmatrix} \tag{2.32}$$

If we apply a block Gaussian eliminate of the unknown X_e, the new system to be solved is defined in terms of the Schur factorisation with a Schur complement defined as $S_c = A_{cc} - A_{ce}A_{ee}^{-1}A_{ec}$. Solving the new equation system (2.33) can be achieved one variable at a time.

$$\begin{pmatrix} I & 0 \\ A_{ce}A_{ee}^{-1} & I \end{pmatrix} \begin{pmatrix} A_{ee} & 0 \\ 0 & S_c \end{pmatrix} \begin{pmatrix} I & A_{ee}^{-1}A_{ec} \\ 0 & I \end{pmatrix} \begin{pmatrix} X_e \\ X_c \end{pmatrix} = \begin{pmatrix} F_e \\ F_c \end{pmatrix} \tag{2.33}$$

Here, static condensation is applied to a hybridised system. In a hybridised system the variable which carries the trace contributions exhibits some global coupling. Therefore, it makes sense to eliminate the other two variables in the system, to solve globally for the trace variable and then recover the other two variables with help of the trace solution. The advantage of this process is that the local recovery can be fully expressed on a local level and no global solve is required. A continuation of example 2.2.1, showing condensation and local recovery, is given in example 2.2.2.

Ex. 2.2.2. *Static condensation of a hybridised problem*

1. In order to solve the hybridised equations system (2.29) - (2.31) static condensation has to be applied and a global system for the trace variable Λ has to be solved. The linear algebra in equations (2.35) and (2.36) local. Note that index 0 corresponds to intra-cell velocity interactions, index 1 to pressure interactions and index 2 to trace velocity interactions.

$$\mathbf{S}\Lambda = \mathbf{E} \quad \text{with } \mathbf{S} = \left\{S^K\right\}_{K\in\Omega_h}, \mathbf{E} = \left\{E^K\right\}_{K\in\Omega_h}, \Lambda = \left\{\Lambda^K\right\}_{K\in\Omega_h} \tag{2.34}$$

$$S^K = A_{22}^K - \left(A_{20}^K A_{21}^K\right) \begin{pmatrix} A_{00}^K & A_{01}^K \\ A_{10}^K & A_{11}^K \end{pmatrix}^{-1} \begin{pmatrix} A_{02}^K \\ A_{12}^K \end{pmatrix} \tag{2.35}$$

$$E^K = F_2^K - \left(A_{20}^K A_{21}^K\right) \begin{pmatrix} A_{00}^K & A_{01}^K \\ A_{10}^K & A_{11}^K \end{pmatrix}^{-1} \begin{pmatrix} F_0^K \\ F_1^K \end{pmatrix} \tag{2.36}$$

2. Let $\mathbf{U} = \left\{U^K\right\}_{K\in\Omega_h}$ and $\mathbf{P} = \left\{P^K\right\}_{K\in\Omega_h}$, then $\mathbf{U}$ and $\mathbf{P}$ can be locally recovered with the solution of equation (2.34). Further, let $S_{1i} = \left(A_{1i}^K - A_{10}^K \left(A_{00}^K\right)^{-1} A_{0i}^K\right)$.

$$S_{11}P^K = F_1^K - A_{10}^K \left(A_{00}^K\right)^{-1} F_0^K - S_{12}\Lambda^K \tag{2.37}$$

$$A_{00}^K U^K = F_0^K - A_{01}^K P^K - A_{02}^K \Lambda^K \tag{2.38}$$

The hybridisation process from breaking the continuity, over Schur complement factorisation up to the local recovery can be encapsulated in a preconditioner [21]. Static condensation can also be used as a standalone preconditioner for systems which are not hybridised but allow similarly for a local elimination [2]. Consider the equation system corresponding to the weak formulation of the hybridised, mixed Poisson problem in equation (2.32) as $\mathbf{Ax} = \mathbf{b}$, where matrix $\mathbf{A}$ consists of 3 blocks and $\mathbf{A}_{i:j}$ denotes the blocks from i to (exclusively) j.

$$\mathbf{P}^{-1} := \begin{pmatrix} \mathbf{I} & -\mathbf{A}_{0:2,0:2}^{-1}\mathbf{A}_{0:2,2} \\ \mathbf{0} & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{A}_{0:2,0:2}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{S}_{\lambda}^{-1} \end{pmatrix} \begin{pmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{A}_{2,0:2}\mathbf{A}_{0:2,0:2}^{-1} & \mathbf{I} \end{pmatrix} \tag{2.39}$$

$$\text{with } \mathbf{S}_{\lambda} := -\mathbf{A}_{2,0:2}\mathbf{A}_{0:2,0:2}^{-1}\mathbf{A}_{0:2,2} \tag{2.40}$$

If static condensation is interpreted as an all-at-once system the corresponding preconditioning operator would be defined as in equation (2.39) and depend on the Schur complement $\mathbf{S}_{\lambda}$.

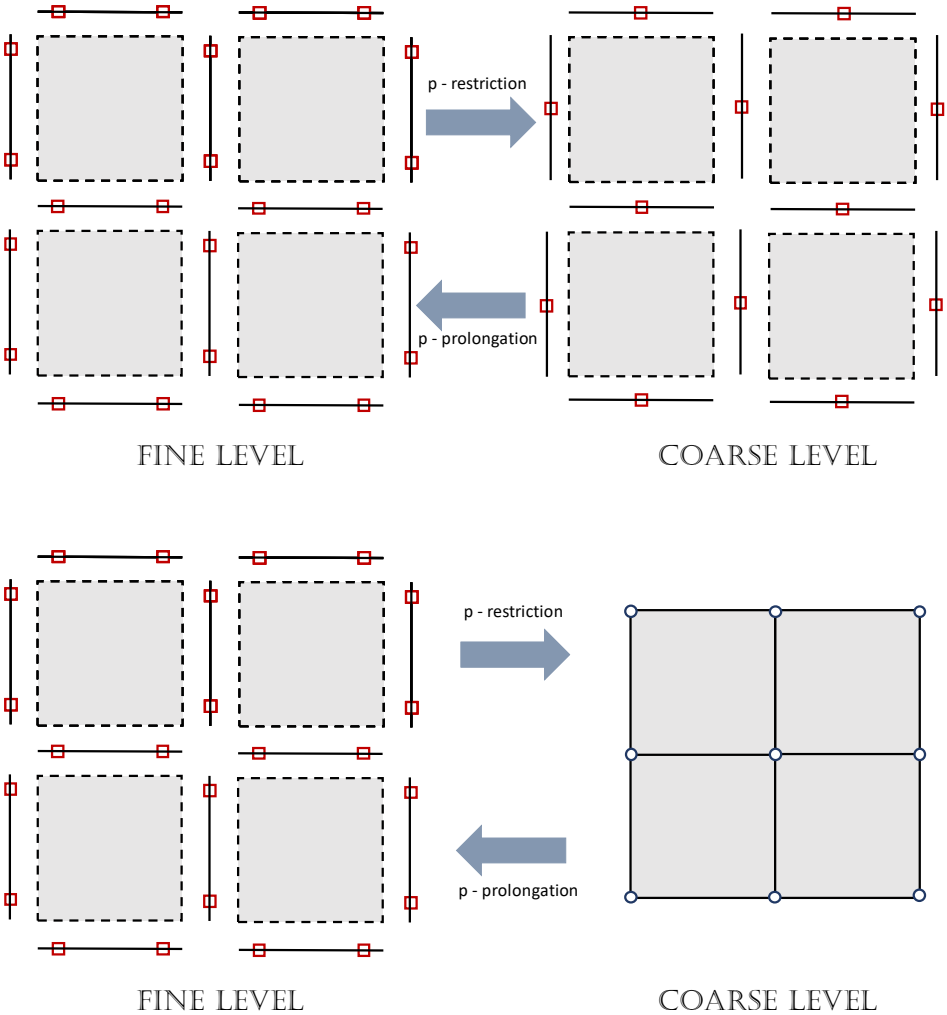


Figure 2.9: Nested (top) vs non-nested (bottom) p-multigrid on a quadrilateral mesh. In this thesis only non-nested p-multigrid is considered.

2.2.4 Gopalakrishnan-Tan multigrid

Multigrid is a well-known solving and preconditioning technique predominantly for elliptic problems. In multigrid algorithms, problems are efficiently solved on a fine scale by considering the problem on a grid hierarchy. The solve on the coarsest grid in the hierarchy is cheaper and presents information about the residual which helps to reduce the error on all length scales on the finer grid. The selling point of multigrid algorithms is their promise of $\mathcal{O}(n)$ complexity per cycle, where n is the number of unknowns.

Multigrid algorithms can be classified into algebraic and geometric methods. In algebraic multigrid, the structure of the system matrix is exploited in order to coarsen the differential operator, whereas in geometric multigrid the information is coming from a rediscretisation of the spaces. Algebraic methods require the explicit assembly of matrices, which becomes prohibitively expensive for high order FEM. Therefore, the focus is set on geometric multigrid algorithms here.

Different geometric multigrid algorithms for FEM can be categorised by

- Construction of the coarse operator (see figure 2.9 for an example)
  - *Nested:* The fine space spans the coarse space.
  - *Non-nested:* The coarse space is not a subspace of the fine space.
- Type of refinement
  - *h-refinement*: The mesh is coarsened.
  - *p-refinement*: The space is coarsened by reducing the polynomial degree of the basis functions.
- Type of cycle: *V-cycle*, *W-cycle* or *F-cycle*, where each cycle specifies a different order of the multigrid building blocks which are restriction, prolongation and smoothing

Gopalakrishnan-Tan multigrid is a non-nested p-multigrid algorithm developed and extended in [49]–[51] and applied to the shallow water equations in [45]. In this thesis, it is used to precondition the iterative solve for the trace system, see e.g. equation (2.34) in the hybridisation preconditioner as explained in chapter 2.2.3. The coarse problem is constructed on a different space than the problem on the fine level. The DOFs defined on the trace space, (the space on the facets of the mesh) on the fine level are moved to a coarser level by using a low order space which is defined on the cells of the mesh. This is visualised in the figure 2.9. The advantage of non-nested multigrid here is that solvers can be used which are already known to be performing well on the coarse space problem. The alternative approach is to coarsen the trace space as presented in [52], [53], but this has the disadvantage of only being scalable with the mesh size under imposing some assumptions like enough smoothness [53].

The details of the solving procedure of a non-nested p-multigrid algorithm for FEM are explained in the following. In order to move the residuals and solutions between the finer and coarser levels, *restriction* and *prolongation* operators are required. In the multigrid algorithm, the fine residual is restricted to a coarse residual to which the coarse solver is applied. The coarse solver produces a coarse solution which is prolonged to a fine solution. On top of that, a *smoother* which removes the high-frequency error at every level has to be defined.

In our application the smoother is a Chebyshev solver with Jacobi preconditioning. For an explanation of the Chebyshev acceleration refer to [44]. The prolongation and restriction operation are defined in [49].

2.3 Code optimisations

2.3.1 Matrix-free methods and Actions

Krylov subspace methods involve solely matrix-vector operations and an assembly of the matrix and vector separately is not strictly required. Matrix-vector products can be replaced by an assembly of a parametrised variational form (Action).

Def. 2.3.1. An Action

An action is an operator, which is applied on a variational form of rank n and a coefficient, and results in a form of rank $n-1$. The reduction in rank stems from replacing one of the arguments of the original form with the coefficient. The input form must take at least one argument.

The reason why it is possible to replace the multiplications with Actions is the bilinearity of the variational forms. If the solvers are applied on local operators as introduced in this thesis, an action on a local level is required. Therefore, let a local matrix-vector multiplication $\mathbf{A}\cdot\tilde{\mathbf{u}}$ be considered, where $u(x) = \sum_{j=0}^{p} \tilde{\mathbf{u}}\Psi_j(x)$. The basis functions Ψ are local basis functions, therefore we call the operation in equation (2.41) a 'local' action.

$$\mathbf{A}\cdot\tilde{\mathbf{u}} = \sum_{j=0}^{p} a(\Psi_j,\Psi_i)\,\tilde{\mathbf{u}} = a\left(\sum_{i=0}^{p} \tilde{\mathbf{u}}\Psi_j(x),\Psi_i\right) = a\,(u,\Psi_i) \tag{2.41}$$

If the linear solver is preconditioned, the *whole* system is sought to be solved in matrix-free manner, so the preconditioner has to have the potential to be matrix-free as well. One possible choice of a matrix-free preconditioner is geometric multigrid. In this thesis, the Gopalakrishnan-Tan multigrid introduced in section 2.2.4, a geometric non-nested p-multigrid algorithm, is considered on the trace system of the hybridisation preconditioner presented in section 2.2.3. The most expensive operation in geometric multigrid is the smoothing operation. Therefore, in particular, the smoother should be considered in a matrix-free form.

Matrix-free methods trade the cost of load/store operations of the operators against the computation of matrix-vector products on the fly. No matrices are saved in matrix-free methods, and the actions (the matrix-free equivalent of a matrix-vector product) need to be recalculated whenever either operator in the product changes. Therefore, matrix-free methods need more FLOPS and ensuring a faster assembly of a 1-form than storing and reusing the result of a matrix-vector product is crucial. For higher-order FEM, this is possible, if sum-factorisation of the elements is supported, see the following chapter 2.3.2. The cost of matrix-free methods utilising sum-factorisation results in a complexity decrease, in particular, the cost of matrix-vector multiplication $\mathcal{O}(p^{2d})$ reduces to the cost $\mathcal{O}(p^{d+1})$ for the matrix-free, sum-factorised Action. p is the polynomial order and d is the dimension.

Matrix-free methods are particularly interesting in parallel in combination with explicit vectorisation for discretisations since only vector operations are needed, see chapter 2.3.3.

2.3.2 Sum factorisation

The idea of sum factorisation has been introduced in [54]. Its core idea is to exploit that FEM on quadrilateral and hexahedral geometries have the special property that the elements can be build through tensor products. This is exploited when building the operators for the variational problem by factorising expressions into products of smaller matrices. A very well-explained example has been published in section 3.2 of [30]. For simplicial elements, sum-factorisation is more complicated than for quadrilateral and hexahedral cells, because the higher order basis functions cannot be simply decomposed into one-dimensional basis functions.

Sum factorisation is advantageous since it decreases the computational complexity of the assembly by a substantial amount. Note that while in matrix-free methods only linear forms have to be assembled, the decrease in complexity e.g. for the local assembly of a linear form on a hexahedral cell still decreases from $\mathcal{O}(p^6)$ to $\mathcal{O}(p^4)$ from a naive to sum-factorised approach [55]. p is the polynomial order.

2.3.3 Vectorisation of FE assembly

Many modern computer architectures can execute **S**ingle **I**nstructions on **M**ultiple **D**ata (SIMD), e.g. graphic processor units (GPUs), vector processors or central processor units (CPUs) with vector extensions. One parallelisation technique that exploits the benefits of this type of architecture is vectorisation. Instead of looping over scalar operations the instructions are executed on vectors of data.

A main limitation for vectorisation of FE assembly is that the local kernels, in particular of low order methods, do not act on large amounts of data. For vectorisation, the data array dimension, i.e. the loop trip, should match the vector lane width of the architecture. The data in single low order FE assembly kernels are smaller than the vector lane extent, however. One approach to overcome this limitation is to use data alignment and padding. This has turned out to lead to inconsistent performance results across different problems [56]. The second approach is to batch the kernel applied to different elements into groups and then parallelly execute the local assembly for the batches. The batching of kernels can be achieved through rewriting the loops in the kernels. Auto-vectorisation algorithms usually fail to vectorise over the element loop because it is not innermost [28].

Thankfully, automatic code generation is a way out of redesigning the generated program and its loops for each hardware design on its own over and over again. The loop transformation software package Loo.py supplies techniques for a compile-time reordering of operations and dependency analysis for various architectures automatically, where the frontend developer is in control of hardware-specific information [57].

In this thesis, a particular interest lies in the vectorisation of local linear algebra operations. A visualisation of how the kernels are vectorised and further details can be found in section 3.3.

2.4 The software framework Firedrake

Firedrake is a tool for translating PDEs, discretised with FEM, into equation systems and automatically solving them with help of the Portable, Extensible Toolkit for Scientific computation (PETSc) [58], [59]. The translation is based on the concept of automatic code generation and the layering of DSLs. While the main code base is written in Python, the code is lowered into C in the translation process, which is beneficial for performance.

A deep understanding of the translation chain in Firedrake is necessary, because of the involved nature of the performance improvements presented in this thesis. For a visual presentation of Firedrake's infrastructure see figure 2.10. Firedrake is well suited for the performance improvements introduced in this thesis since many of the building blocks which are needed for them are readily available. In particular, the support of the required compatible FE spaces, the hybridisation preconditioner, the Gopalakrishnan-Tan preconditioner, tensor product elements and sum-factorisation, and vectorisation for expressions which do not involve local linear algebra operations, were implemented by former students. The mathematical and computer scientific arguments for the expected performance improvements are generic, however, and all ideas can theoretically be integrated with other Finite element software packages.

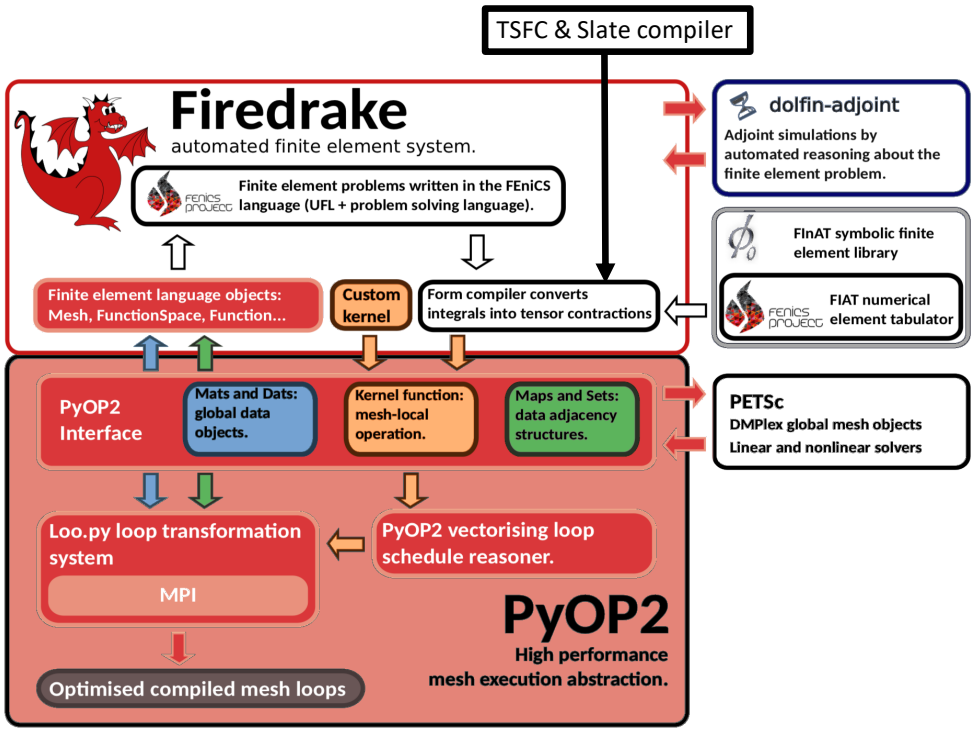


Figure 2.10: Components of Firedrake and their interaction with PyOP2, dolfin-adjoint and PETSc

In Firedrake, in the first layer of abstraction, the user formulates a PDE in 'FEM language' represented by the Unified Form Language (UFL) [60]. Sometimes the user wants to introduce complicated local linear algebra operations on top of their problem definition. This can be useful for example to precondition the equation systems in order to reduce some resources (e.g. time) for solving them. The operators and operations to specify the linear algebra on top of the UFL language are provided by the System for Linear Algebra on Tensor Expressions (Slate) [61]. In this thesis Slate is used to express the hybridisation preconditioner which was introduced in section 2.2.3.

The interface underneath the first layer of abstraction, which is usually not accessed by Firedrake users, lowers some of the FEM specific structures of the problems defined in UFL/Slate. The interface consists of the Two-stage Form Compiler (TSFC) [55] for UFL and the Slate Compiler (Slac)[61]. The compilers provide translations of the frontend problem (FE variational forms and linear algebra operations) into local kernels. From a computer science point of view, the term 'local kernels' in Firedrake refers to C functions which are executed on one part of the problem. From a mathematical point of view, the local kernels are responsible for the calculations which fill sub-equation systems with local contributions (local assembly), see section 2.1.3. The local (assembly) kernels help to approximate the solutions on a single element of the mesh. This layer is the one which is discussed the most throughout the thesis.

In the penultimate layer, for translation of the domain specifications, the local kernels of the previous interface are fed into a High-level Framework for Performance-portable Simulations on Unstructured Meshes (PyOP2) [62]. PyOP2 executes the kernels in parallel over data stored on a mesh, gathering the local contributions of each element of the domain in a big system. This is called global assembly, refer to section 2.1.3. Once the operators and coefficients are globally assembled, the final equation systems are passed to PETSc, which solves the systems and returns the approximate solution.

Chapter 3

An automated framework for vectorised linear algebra on Finite element tensors

One of the key benefits of Firedrake is that it allows expressing the problem at a high abstraction level and is therefore highly flexible. All of Firedrake's components are developed under two over-arching goals besides the flexibility at top-level. The first goal is to achieve greater portability for solving FE problems across computer platforms. The second is solving the FE problems in a highly performant manner. In this chapter, a new component is introduced to Firedrake, a compiler which translates one of Firedrake's DSLs Slate (see chapter 3.1.1) to Loo.py (see chapter 3.1.3) with an intermediate representation, the Tensor Algebra Language (GEM) (see chapter 3.1.2). Similar to other components in Firedrake, the new compiler is necessary to improve performance and performance portability.

In the development of one of the earliest components of Firedrake, greater performance ability was achieved by replacing the DOLFIN interface for finite element global assembly in FEniCS with PyOP2 [62]. PyOP2 could not only generate code for CPU, but also for GPU architectures for the first time. Unfortunately, the global assembly strategy on GPUs did not provide reliable performance due to a lack of support of assembling PETSc matrices on GPUs. Therefore, calculations on GPUs were only available in limited form. This is changing now for multiple reasons, so that a broader support on GPUs can be reintroduced again. 1) Only vectors are globally assembled in matrix-free methods. Furthermore, matrix-free preconditioners are now supported in Firedrake as well. 2) Support for matrix assembly on GPUs has recently been added to PETSc. 3) The Loo.py language supports code generation on various platforms, including GPUs. The first reason for the introduction of a new compiler, which translates Slate to Loo.py, is that it flattens the path to GPU support for solving finite element problems with preconditioners based on local linear algebra operations. The actual code generation for GPUs is beyond scope in this thesis, however.

PyOP2 achieved the second arching goal, high performance, by using MPI-parallelism. Further optimisations on the local kernels in Firedrake have been achieved through a Compiler For Fast Expression Evaluation (COFFEE) [56] and TSFC [55]. In code generation software, the potential of code optimisations lies in the information inherent to the problem, which general purpose compilers, e.g. the Intel compiler, are incapable of exploiting. Exploiting more information at an earlier stage in the compiler pipeline can help to do smarter code synthesis.

One of the code optimisations introduced in COFFEE was code vectorisation. The code vectorisation of the local kernels was achieved via padding and data alignment [56]. This turned out to not to deliver robust performance. Structure exploitation at a higher level, in form of batching the loops over multiple elements in the iteration set of the PyOP2 kernels, delivers more consistent performance results, which was proven in [28]. Henceforth, this kind of vectorisation is called cross-element vectorisation. The loop transformation DSL Loo.py [57] delivers an automated way to perform the required code transformations for the batching. This is the main reason for the introduction of a new compiler, which translates Slate to Loo.py. The new compiler uses the intermediate tensor language GEM such that existing compiler technology in TSFC can be reused for the code generation of the Slate kernels.

3.1 The DSLs Slate, GEM and loo.py and their compilers

3.1.1 Slate and Slac

Slate [2] is a DSL which allows its users to define local dense linear algebra operations on tensors assembled from local Finite element forms. The language allows an easy expression of local preconditioners. In the Firedrake framework, Slate is for example used for static condensation and hybridisation, see section 2.2.3. It is important to understand that the Slate tensors are built through *local* assembly, see section 2.1.3, of the variational forms. All linear algebra operations on the tensors are therefore local too.

Slate's implementation builds on top of UFL (and TSFC) on the highest level of abstraction of Firedrake. Its original compiler, Slac, accesses TSFC technology to fill the local tensors with data, but has its own translation to transform Slate operations into code. In particular, Slac translates Slate into constructs from the Eigen package [63]. Eigen provides C++ linear algebra functions optimised for small dense tensors. The contributions of the local algebra operations are assembled into the global matrix by PyOP2, similar to the TSFC kernels.

Unary			Binary		
Inversion:	T_1^{-1}	(`Inverse`)	Addition:	$T_1 + T_2$	(`Add`)
Transposition:	T_1^T	(`Transpose`)	Multiplication:	$T_1 \cdot T_2$ and $T_1 \cdot C$	(`Mul`)
Negation:	$-T_1$	(`Negative`)	Solving:	$T_1 \backslash C$	(`Solve`)
Indexing:	$T_1[i_0, i_1]$	(`Block`)			

Table 3.1: Original unary and binary operators in the Slate language. The names of the Slate nodes are specified in brackets.

Let $V(F)$ be a vector space on the field F, then the variational forms describing a FE problem, are k-linear maps $f_k : \bigotimes_k V_k(F) \to F$. At the Slate level in the Firedrake pipeline, a tesselation $\mathcal{T}_h$ with cells $K \in \mathcal{T}_h$ has been chosen, as well as a discretisation V_h for the vector space $V(K)$, see chapter 2.1.2. While tensors are generally basis-independent objects, here the tensors in Slate are more specifically speaking multidimensional arrays, since a basis has been chosen.

Denote each tensor belonging to one FE problem by $T_i = T_{k_i,i} = \bigotimes_k V_h \to W$ and each coefficient in V_h by C_i. Let the set of m_1 tensors and m_2 coefficients be $\mathbf{T} = \{T_0, \ldots, T_{m_1-1}, C_0, \ldots, C_{m_2-1}\}$. The tensors can be scalar, vectors or matrices depending on the forms. In Slate, the terminal tensor nodes are named `Tensor`, the coefficient nodes are named `AssembledVector`. The original set of tensor operations in Slate is $\mathbf{F} = \{+, \backslash, \cdot, ^{-1}, ^T, -\}$ and the operations can be classified into being unary and binary, see table 3.1. For a detailed description of the Slate language refer to [2].

Slate expressions built from **T** and **F** are internally represented as directed acyclic trees (DAGs). In the DAGs of Slate expressions `Tensor` and the `AssembledVector` are always the leaf nodes.

Ex. 3.1.1. A DAG for the Slate expression $T_1 \cdot T_2 \cdot T_3 \cdot C$

Let $T_1, T_2, T_3, C \in \mathbf{T}$, where T_i are tensors and C a coefficient in V_h. Consider a variational form as presented in example 2.1.1 on a local level. A corresponding tensor to this local form would be defined as `T1 = Tensor(a)`. The corresponding DAG to a Slate expression `Mul(T1, Mul(T2, Mul(T3, C))` is visualised in figure 3.1.

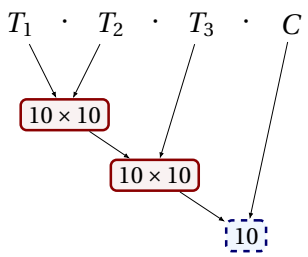


Figure 3.1: A Slate DAG. Red boxes with continous lines represent matrix-valued temporaries, blue boxes with dashed lines represent vector-valued temporaries. Values in boxes show shape information. Shape compatabilities are checked internally in the Slate compiler.

One of the key feature of Slate is that the language allows one to execute expensive operations like inverting a tensor or solving a system similar to $T_1 x = C_1$ on a local level, where T_1 is a matrix and C_1 a coefficient. These linear algebra operations are more complex than others and need special attention by the compiler, which is explained section 3.2. There is another part of the Slate infrastructure, the `Block` node, which is used to index into tensors and has a more involved code generation. Its complexity becomes apparent in its translation to lower levels in the pipeline and in its optimisations. I developed a new optimisation pass for Slate expressions involving blocks as pointed out in a later chapter, see section 4.2.1.3.

3.1.2 TSFC, GEM and Impero

In its early years Firedrake used a modified version of FEniCS Form Compiler (FFC) [64] in order to lower the FE weak form, as defined by the user in UFL, to local assembly kernels written in C. The UFL forms were directly translated into the Unified form-assembly code (UFC) [65], lacking an intermediate representation. This was changed with an introduction of the tensor notation intermediate language GEM as part of the new Two Stage Form Compiler (TSFC) [55]. The intermediate language comes with the advantage that code optimisations can be employed at an earlier stage, where FE specific information has not yet been lost. Employing code optimisations earlier means that they can be applied at a higher abstraction level. This is key for the success of Firedrake.

With TSFC, a UFL form is translated into the tensor notation language GEM first. Then, GEM is translated into Impero, which is a language in "an abstract syntax tree format that helps the generation of imperative code" [55]. In contrast to GEM, Impero is scheduled. Impero enforces a partial loop ordering. Impero is further translated to C (via Loo.py). GEM is based on a similar tensor notation language as is employed in FEniCS, but with less constraints on the free indices. The code optimisations in the first development phase of TSFC were inlining, loop hoisting, loop fusion, topological ordering with an adapted Kahn algorithm, constant folding of zero tensors, fast Jacobian evaluation for affine cells, unrolling short index sums and some of the optimisations provided by COFFEE [55].

After Firedrake's expansion with a smarter library for finite elements, called Finite Element not Automated Tabulator (FInAT) [30], a new set of optimisations could be provided through TSFC. While the basis functions in the former library of finite elements, called Finite Element Automated Tabulator (FIAT), are tabulations, structure is exploited to avoid the tabulations in FInAT. FInAT constructs structure-preserving representations of finite elements on tensor product cells, and some vector- and tensor-valued finite elements. The kernels built on tensor-product cells are optimised through sum factorisation and delta cancellation [55]. These optimisations are particularly useful for high-order matrix-free methods, see chapter 2.3.1. Sum-factorisation and delta cancellation are implemented as transformations of tensor expressions in the GEM language.

GEM [55] is a language, which allows tensors to be represented in Einstein notation. Einstein notation provides a natural way of expressing tensor contractions. GEM comes into play at a stage in Firedrake, where the continuous variational problem has been turned into a discrete problem and finite element specific and geometric information have already been lowered into purely algebraic expressions. An example for a contraction at GEM level would be $\sum_j \mathbf{b}[j]\,\mathbf{c}[j]$ where **b** and **c** are vector-valued. In Einstein notation that contraction is written as $b_j c_j$. In this notation, summation is implicitly assumed over indices which occur twice in the same product.

In GEM, a *shape* characterises the dimensions of a tensor, whereas so called *free indices* are unrolled shapes. The free indices may refer to several entries in the tensor depending on their extent. The terminal nodes in GEM are `Literal` and `Variable`, where `Literal` is scalar-valued and reserved for constant nodes. `Variable` is a placeholder for something that has a shape. Another terminal node is `Index`. A table of the original GEM operations which are relevant in the context of the thesis are specified in table 9.1.

Operations on shaped objects			**Operations on scalars**		
Indexing:	$T_1[i]$	(`Indexed`)	Addition:	$t_1 + t_2$	(`Sum`)
	T_1[slice]	(`FlexiblyIndexed`)	Multiplication:	$t_1 \cdot t_2$	(`Product`)
			Einstein sum:	$\sum_i t_{1,i}$	(`IndexSum`)
			Pull-up:	$]t_{1,i}[_i$	(`ComponentTensor`)

Table 3.2: Operations in the GEM language. Capital symbols denote objects with shape, lower-case symbols denote scalars. The names of the nodes in GEM are specified in brackets.

Slate and GEM both represent terminal tensors as a node **A**. Terminal nodes do not have children, they either carry data or are placeholders which are filled with data later. In Slate the node is called a `Tensor` and in GEM it is a `Variable`. At which level the operations on the tensors are applied differs between the languages, however. While Slate operates on the whole tensor, GEM operations access the tensor entry by entry. In GEM, operations on the shape objects always need to be represented with an operation which turns a scalar expression into a shaped object considering the free indices, which I call a pull-up. Mathematically this operation is represented by $]A_{ij}[_{ij}$, where A is the scalar object, the free indices are i, j and the result of the operations is a shaped object. A matrix addition would e.g. be expressed as $]A_{ij} + B_{ij}[_{ij}$. The corresponding GEM expression would be `ComponentTensor(expression`$_1$`, multiindex)` with `expression`$_1$ = `Add`(a, b)). Here, a and b are indexed tensors. The nodes a and b in the example are indexed expressions defined as `Indexed`(A, `multiindex`), and `Indexed`(B, `multiindex`), where A and B are of type `Variable`.

Indexing is the opposite operation to 'pulling up' tensors. In GEM indexing tensors is represented through `Indexed(expression`$_2$`, multiindex)`. The node expression$_2$ must have a `shape`, but expression$_1$ does not. Structurally, the main difference between the index notation in UFL and GEM is a loosening of the constraints on the free indices so more optimisations are possible.

3.1.3 Loo.py and pymbolic

Loo.py is a code generator for CPU- and GPU-type shared memory systems, which is embedded in Python and based on the polyhedral model [57]. Loo.py makes it possible to invoke transformations needed for additional optimisations like loop tiling at a high level of abstraction. Sets and relations for the polyhedra are expressed via the ISL library [66]. Loo.py is designed with a strong separation of concerns, similar to Firedrake, between loop polyhedra plus partially ordered instructions and the transformations on them. An example of Loo.py kernels are shown later in section 3.2.3.

The main problem with the vectorisation in Firedrake as it was realised in COFFEE was performance inconsistency across different problems due to missed opportunities to vectorise. In COFFEE, vectorisation is implemented with data padding to match inner loop trips with the length of the vector unit. Modern SIMD architectures tend to have wide SIMD units, where the cost of data padding may exceed its benefit [28]. Moreover, hoisted loops as they typically arise from sum factorisation in TSFC are non-perfect loop nests, which are difficult to vectorise with data padding [28].

Better performance can be achieved by vectorising over the outer loop, the loop over the elements of the mesh. Then, local kernels can be batched into groups, where the total length matches the SIMD unit length and each element kernel is calculated on its own vector lane. The DSL Loo.py provides an automation of sequences of transformations based on the polyhedral compiler model, which are needed to achieve vectorisation for batched kernels. The transformations required are explained in section 3.3. For the vectorisation of FE kernels coming from UFL formulation, loop fusion of outer and inner kernel, such that the global kernel contains all the information it needs for inter-element vectorisation, is achieved in [28]. Vectorisation itself is employed with Loo.py via pragmas and vector extensions. Performance results for cross-element vectorisation indicated more consistently the efficacy of vectorisation [28].

In [28] FE problems without local linear algebra operations are vectorised when possible, but it required a translation to Loo.py in TSFC and PyOP2 before further translation to C. Slate as developed in [61] had not been ported to translate to Loo.py before the work in this thesis, which prohibited its vectorisation.

3.1.3.1 Loo.py kernels

The main features describing a kernel in Loo.py are given in the following. For a more extensive description visit the Loo.py documentation [67]. An example of how these transformations can be used for high order FEM is given in [68].

1. The **name** of the kernel is user-defined.
2. Iteration domains are described by loop variables, which are called **inames**, and loop bounds are enforced through constraints in ISL syntax. Inames can be tagged for example such that the loop is unrolled.
3. **Instructions** have to be scalar or subscripted array assignments. They are mainly characterised by an ID, the expression, dependencies to other instructions and on inames. The expression can be a call to a function (`CallInstruction`), an `Assignment` or an instruction expressed in C (`CInstruction`). Loo.py expressions are wrappers of pymbolic [69] expressions.
4. The **arguments** are input parameters of the kernels. They are mostly defined via name, shape and type, but also allow to control physical storage layout via `dim_tags` and address space keywords.
5. **Temporary variables** are similar to the kernel arguments, but correspond to private and local address spaces of the kernel.

3.1.3.2 Loo.py transformations

Loo.py introduces the transformation on the polyhedral data model by using Python pragma-like constructs for example via tags on data objects.

1. **Splitting** of inames: for example for loop tiling
2. **Tagging** iname implementations: for example for loop unrolling, instruction-level parallelism, vectorisation
3. **Data layout transformations**: for example for data padding
4. **Prefetching** of substitution rules: treating storage for parallelism

3.2 TSSLAC: A Two Stage Slate compiler

Here, I introduce a new Slate compiler, the Two Stage Linear Algebra Compiler (TSSLAC), which replaces the previous compiler, Slac. Originally, the compiler Slac generated C++ code with help of Eigen. The flaws of Slac are a) that it produces kernels which cannot be vectorised with existing Firedrake technologies and b) the Slate kernels are not optimised in any way. In order to overcome Slac's flaws I have written the compiler TSSLAC, which translates Slate into Loo.py with an intermediate step over GEM. Therefore, TSSLAC exploits more domain induced structure in code optimisations and introduces more efficient/performance portable vectorisation.

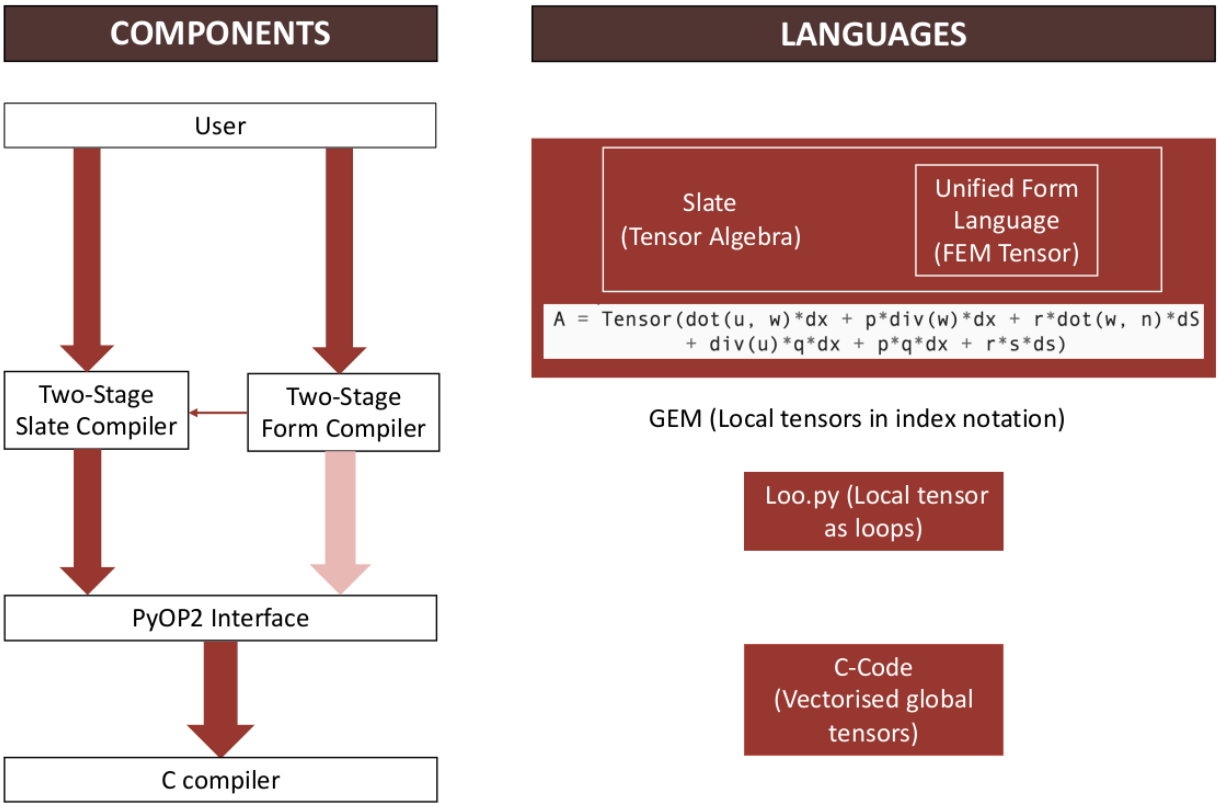


Figure 3.2: Firedrake compiler stages with TSSLAC

The new Slate compilation, like the form compilation with TSFC, can be seperated into two stages. The first stage is a translation of the Slate expressions into GEM expression, the second stage is a translation from GEM to Loo.py. After these two stages the Slate expression kernel and the local assembly kernels, generated by TSFC, have to be connected to each other, and further passed to PyOP2. A summary of the algorithm is presented in figure 1. The input parameters to the Slate compilation are a Slate expression and compiler parameters which for example determine if the expression should be optimised or not. The critical translations from Slate to GEM and GEM to loopy are explained in section 3.2.1 and 3.2.2. Additional optimisations on Slate expressions are explained in a later chapter as part of the new local matrix-free infrastructure, see section 4.2.1.

Algorithm 1: Slate compilation

Data: Slate expression, compiler parameters

Result: PyOP2 local kernel

```
1  if "optimise" in compiler parameters then
2      push blocks inside the Slate expression;
3      push diagonals inside the Slate expression;
4      push multiplications inside the Slate expression;
5      remove double transposes;
6  translate Slate expression to GEM expression;
7  translate GEM expression to imperoC;
8  translate ImperoC to Loo.py kernel;
9  merge Slate expression kernels and local assembly Loo.py kernels;
10 register LAPACK inverse and solve Callables ;
11 embedd merged Loo.py kernel into PyOP2 local kernel;
```

3.2.1 Stage 1: Slate to GEM

In the first critical stage, Slate is translated into GEM. Translating to GEM allows a reutilisation of compiler optimisations from TSFC e.g. constant folding for scalars, hoisting of the evaluation of all cellwise constant subexpressions out of the quadrature loop, fast Jacobian evaluation for affine cells, delta cancellation, sum factorisation and loop unrolling [55]. All linear algebra operations in Slate and their translations to GEM are presented in table 9.1

All simple linear algebra operations in the Slate language could be translated into already existing GEM nodes. For the slightly more involved linear algebra operations I introduced `Inverse` and `Solve` nodes in GEM, which are like `Variable` just placeholders for something which has a shape, but need special treatment further down in the compiler stack when GEM is translated to Loo.py.

A list of translations from Slate operations into GEM constructs is given in the following. Let A and B denote matrices and b a vector induced by a bilinear form a and a coefficient f. The terminal tensors of Slate are translated into placeholder variables. Slate's `Inverse` and `Solve` nodes are replaced by their GEM counterparts. A Slate multiplication translates into a tensor contraction using `IndexSum`.

Transposes are built with a `ComponentTensor` with exchanged indices. Negations are constructed with a `ComponentTensor` over -1 literals and a multiplication. Parts of mixed systems can be accessed via `Blocks`, each single block corresponding to different subspaces. The `Blocks` operation is translated to GEM with help of a `ComponentTensor` over a `FlexiblyIndexed` node. The slices to the `FlexiblyIndexed` node contain information about where in the matrix the block is and what its dimensions are.

The tensors are filled with data through local assembly kernels generated by TSFC. At this stage in the code generation, the GEM expression DAG for the linear algebra operations is standalone, using placeholders for the tensors. The GEM expression tree for the tensors is generated separately. Both GEM expressions are separately translated into Loo.py, see section 3.2.2 and the TSFC assembly call is merged into the Slate kernel afterwards.

Slate nodes	**GEM translations of Slate nodes**
`Tensor(`a`)` `AssembledVector(`f`)`	`Variable(name, shape)`
`Inverse(A)` `Solve(A)`	`Inverse(A)` `Solve(A)`
`Transpose(A)`	`ComponentTensor(Indexed(A, indices), tuple(indices[::-1]))`
`Negative(A)`	`ComponentTensor(Product(Literal(-1),` `Indexed(A, indices)), indices)`
`Add(A, B)`	`ComponentTensor(Sum(Indexed(A, indices),` `Indexed(B, indices)),indices)`
`Mul(A, B)`	`ComponentTensor(IndexSum(Product(Indexed(A, tuple(i + [k])),` `Indexed(B, tuple([k] + j))), (k, )), tuple(i + j))`
`A.blocks[number]`	`view(A, *slices)`

Table 3.3: Translations from Slate to GEM. Capital A, B denote tensors, lower-case a a variational form.

```
class INVCallable(LACallable):
    def generate_preambles(self, target):
        assert isinstance(target, loopy.CTarget)

        inverse_preamble =
        """
            #define Inverse_HPP
            #define BUF_SIZE 30
            static PetscBLASInt ipiv_buffer[BUF_SIZE];
            static PetscScalar work_buffer[BUF_SIZE*BUF_SIZE];

            /* C inverse function */
            static void inverse(PetscScalar* __restrict__ Aout, \
                                const PetscScalar* __restrict__ A, \
                                const PetscBLASInt N){


                /* Declarations */
                /* malloc is only used when matrices becomes too big */
                PetscBLASInt info;
                PetscBLASInt *ipiv = (N <= BUF_SIZE ? ipiv_buffer : \
                                      malloc(N*sizeof(*ipiv)));
                PetscScalar *Awork = (N <= BUF_SIZE ? work_buffer : \
                                      malloc(N*N*sizeof(*Awork)));
                memcpy(Aout,A,N*N*sizeof(PetscScalar));


                /* Factorisation */
                LAPACKgetrf_(&N,&N,Aout,&N,ipiv,&info);

                /* Inverse */
                if(info == 0){
                    LAPACKgetri_(&N,Aout,&N,ipiv,Awork,&N,&info);}

                /* Don't fail silent */
                if(info != 0){
                    fprintf(stderr,\"Getri throws nonzero info.\");
                    abort();}

                if (N > BUF_SIZE) {
                    free(Awork);
                    free(ipiv);}
            }
        """
        yield ("inverse", \
              "#include <petscsys.h>\n#include <petscblaslapack.h>\n" + \
              inverse_preamble)
```

Figure 3.3: A Loo.py inverse callable, which is a holder of C code that has callbacks to LAPACK functions.

3.2.2 Stage 2: GEM to Loo.py

The translation from GEM to Loo.py is handled by already existing TSFC technology besides for the inverse and solve nodes. First, ImperoC [34] code is generated from GEM via TSFC, which substantiates loop structures and enforces a partial ordering on assignments. I reuse already existing TSFC functionality for the generation from Impero to Loo.py. The infrastructure was introduced for an experimental vectorisation study with Loo.py in [28].

The new inverse and solve GEM nodes are translated to calls to external LAPACK function calls [70] in the locally matrix-explicit case. The translations for the locally matrix-free cases are covered in chapter 4. LAPACK uses Basic Linear Algebra Subprograms (BLAS) to perform matrix-vector operations needed for the inverse and solve calculations. In figure 3.3, an example is given for the `Callable` of an inverse. Previously, `ComponentTensors` could always be eliminated through TSFC optimisations. For the Inverse and Solve nodes, this is not the case, such that a Loo.py translation for the `ComponentTensor` in GEM had to be provided as well.

3.2.3 Example of a Loo.py Slate kernel

Consider the example of a hybridised mixed Poisson problem, see example 2.2.1. The system can be solved with static condensation as explained in section 2.2.3.2. Consider the last expression in the static condensation procedure, equation (2.38), which represents the local recovery of one of the variables. The expression of interest is $A^K_{00}U^K = F^K_0 - A^K_{01}P^K - A^K_{02}\Lambda^K$. Translated into the Slate language the expression is $A[0,0].\texttt{solve}(F[0] - A[0,1]\cdot\texttt{AssembledVector}(p) - A[0,2]\cdot\texttt{lambda})$, where A is a `Tensor` on the left hand side of the hybridised form of the equations in example 2.2.1, and F the right hand side. The corresponding Loo.py kernels are shown in figure 3.4 and 3.5.

3.3 Vectorisation of Slate kernels

Vectorisation is a parallelism technique where one instruction processes multiple data elements simultaneously. Modern computer architectures make this possible through vector registers which can fit more than one double. The vectorisation of Slate kernels helps to achieve a higher amount of peak performance.

In a code generation software like Firedrake it is important to achieve good performance across problems and computer architectures. The auto-vectorisation implemented in compilers like gcc does currently not deliver good performance. Robust vectorisation of finite element operations is best achieved by vectorising over a loop that is not the innermost loop [28]. This typically defeats autovectorising algorithms, which are usually only capable of vectorising innermost loops.

```
1  KERNEL: slate_wrapper
2  ------------------------------------------------------------------------
3  ARGUMENTS:
4  cell_facets: type: np:dtype('int8'), shape: (3, 2), ...
5  coords: type: np:dtype('float64'), shape: (6), ...
6  output: type: np:dtype('float64'), shape: (3), ...
7  w_0: type: np:dtype('float64'), shape: (3), ...
8  w_1: type: np:dtype('float64'), shape: (1), ...
9  w_2: type: np:dtype('float64'), shape: (3), ...
10 DOMAINS: ...
11 INAME TAGS: ...
12 TEMPORARIES: ...
13 ------------------------------------------------------------------------
14 INSTRUCTIONS:
15 ... initialisations ...
16
17 [id_8,id_9]: T0[id_8, id_9] =
18     subkernel0_cell_to__cell_integral(
19         [i,i_0]: T0[i, i_0],
20         [id_10]: coords[id_10])                                  {id=i0}
21
22 [id_11,id_12]: T2[id_11, id_12] =
23     subkernel1_cell_to__cell_integral(
24         [i_1,i_2]: T2[i_1, i_2],
25         [id_13]: coords[id_13])                                  {id=i1}
26
27 for id_17
28     if cell_facets[id_17, 0] == 1
29         [id_14,id_15]: T4[id_14, id_15] =
30             subkernel2_interior_facet_to__exterior_facet_integral(
31                 [i_3,i_4]: T4[i_3, i_4],
32                 [id_16]: coords[id_16],
33                 [id_18]: facet_array[id_18 + id_17])             {id=i2}
34     end
35 end id_17
36
37 [id_28]: output[id_28] =
38     slate_loopy(
39         [i_5]: output[i_5],
40         [id_19,id_20]: T0[id_19, id_20],
41         [id_21]: T1[id_21],
42         [id_22,id_23]: T2[id_22, id_23],
43         [id_24]: T3[id_24],
44         [id_25,id_26]: T4[id_25, id_26],
45         [id_27]: T5[id_27])                                      {id=i3}
```

Figure 3.4: A Slate wrapper kernel fills the tensors with data from TSFC for equation (2.38)
Line 1: The kernel name
Line 3-9: Arguments passed into the kernel
Line 10: Domains of the loop indices
Line 11: Special tags of the loop indices
Line 12: Temporaries used in the kernel
Line 17-33: Local assembly calls called 'subkernel*' for cell integrals and facet integrals
Line 27-45: Call to kernel for linear algebra operations on the temporaries T0, T2 and T4

```
1  KERNEL: slate_loopy
2  ---------------------------------------------------------------------------
3  ARGUMENTS:
4  T0: type: np:dtype('float64'), shape: (3, 3), ...
5  T1: type: np:dtype('float64'), shape: (3), ...
6  T2: type: np:dtype('float64'), shape: (3, 1), ...
7  T3: type: np:dtype('float64'), shape: (1), ...
8  T4: type: np:dtype('float64'), shape: (3, 3), ...
9  T5: type: np:dtype('float64'), shape: (3), ...
10 output: type: np:dtype('float64'), shape: (3), ...
11 ---------------------------------------------------------------------------
12 DOMAINS: ...
13 INAME TAGS: ...
14 TEMPORARIES: ...
15 ---------------------------------------------------------------------------
16 INSTRUCTIONS:
17 for i
18     t0[i] = 0.0                                                    {id=i0}
19     for i_0
20         t0[i] = t0[i]+T4[i_0+i//3, i+(-3)*(i//3)]*T5[i_0]   {id=i1}
21     end i, i_0
22 for i_1
23     t1[i_1] = (-1.0)*t0[i_1]+T1[i_1]+(-1.0)*T2[i_1, 0]*T3[0]      {id=i2}
24 end i_1
25 [i_2]: t2[i_2] = solve([i_3,i_4]: T0[i_3+i_4//3, i_4+(-3)*(i_4//3)],
26                        [i_5]: t1[i_5])                             {id=i3}
27 for i_6
28     output[i_6] = output[i_6]+t2[i_6]                              {id=i4}
29 end i_6
```

Figure 3.5: A Slate kernel executes the linear algebra operations on the tensors for equation 2.38. The following description projects the code back to the equation.
Line 1: The kernel name
Line 3-10: Arguments passed into the kernel
Line 12: Domains of the loop indices
Line 13: Special tags of the loop indices
Line 14: Temporaries used in the kernel
Line 17-21: $A^K_{02}\Lambda^K$
Line 17-21: $-A^K_{02}\Lambda^K + F^K_0 - A^K_{01}P^K$
Line 25-25: $A^K_{00}.\text{solve}(-A^K_{02}\Lambda^K + F^K_0 - A^K_{01}P^K)$

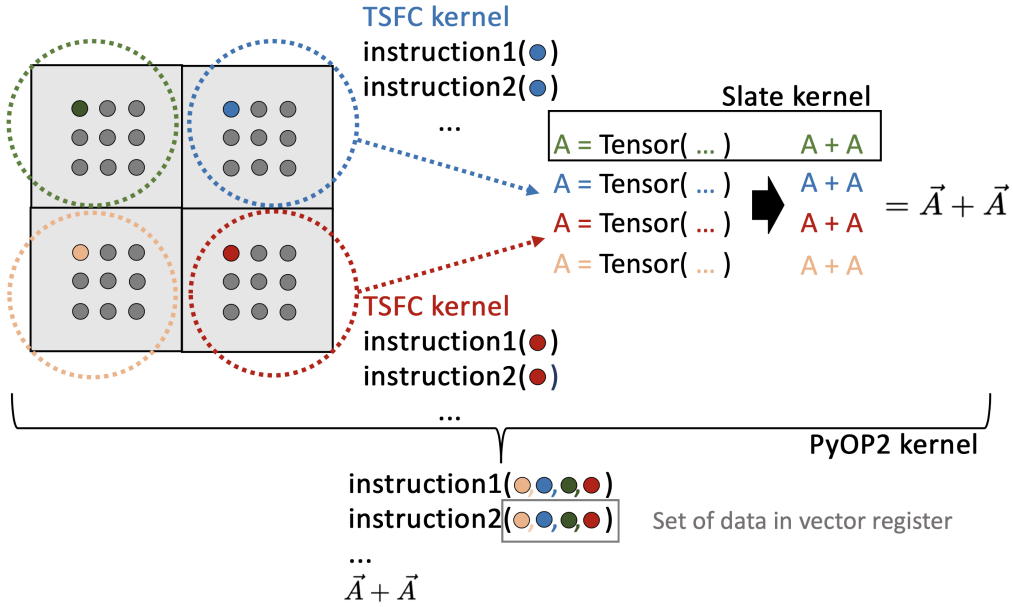


Figure 3.6: Cross-element vectorisation of Slate kernels. TSFC kernels calculate the contributions per cell. Slate kernels access the data from TSFC kernels and calculate linear algebra operations on those contributions per cell. In PyOP2, TSFC and Slate kernels are rewritten so that each instruction is calculated on multiple cells at the same time.

In the vectorisation strategy implemented in Firedrake in [28], instructions in the local and global assembly are executed for multiple finite elements in the mesh at the same time. The vectorisation of Slate kernels presented here uses the work from [28] for global and local assembly, but local linear algebra operations are vectorised too. A visualisation of cross-element vectorisation of Slate kernels can be found in figure 3.6. Cross-element vectorisation has been proven to deliver more consistent speedup in [28], than intra-kernel vectorisation as it was implemented in [56].

3.3.1 PyOP2 vectorisation algorithm

Translating the frontend DSLs in Firedrake to a Loo.py backend gives an easy access to loop manipulations. The result is that very few lines of code are required in PyOP2 to vectorise FE kernels. The vectorisation algorithm to implement the vectorisation in PyOP2 can be found in figure 2. All instructions in the algorithm are a one-to-one Loo.py-to-pseudocode translations. The required changes in PyOP2 amounts to less than 50 lines.

There are two main steps in the PyOP2 vectorisation algorithm. First, the outer loop is split into batches of however many doubles fit the vector registers of the architecture and the loop which should be vectorised is tagged accordingly. Secondly, the format of all variables is changed to hold vectors of data.

Whenever possible a loop over an instruction of scalars turns into an instruction on variables of a type which hold multiple data elements (vectorextension), see [71]. If a variable has type double and the register can fit four doubles, the new variable with vectorextended type can hold four doubles. If vectorisation is not possible (e.g. because there are call to math functions like `abs` which cannot operate on vectorextended variables), Loo.py falls back to surround the loop with an OpenMP SIMD pragma, which gives a directive to the compiler to vectorise it.

In contrast to the approach in [28], the way vectorisation is achieved is not specified through separate vectorextension and OpenMP backend code targets, but it is controlled through loop tags. The study in [28] showed more consistent performance results for vectorisation with help of vectorextensions compared to OpenMP pragmas. Therefore, now there is only one general C vectorisation target where vectorextensions are the default and Loo.py only falls back to OpenMP pragmas if necessary.

There are some technical details to be considered in the vectorisation of FE kernels. The vector register length of the architecture is determined automatically if possible, otherwise it defaults to fit 4 doubles and can be changed by the frontend user through a PyOP2 environment variable. Further, it should be noted that the outer loop is not necessarily cleanly divisible through the amount of doubles which fit in a vector register. When this is the case, the vectorised kernel generated by Loo.py contains a main body for the vectorisation instructions and so called slabs which execute the instructions unvectorised for the modulo iterations. Since the outer most loop bounds are shifted to start from 0 in line 6 of algorithm 2, the generated kernels would only contain final slabs.

There are some constraints on which kernels and instructions cannot be vectorised. The kernels which cannot be vectorised are the following. Conditionals are explicitly vectorised through Loo.py.

- Kernels which assemble matrices[1]
- Kernels which use complex types
- Kernels with read-write access arguments

Single instructions in otherwise vectorisable kernels according to the list above can be non-vectorisable.

- Instructions which are outside the loop which was split
- Instructions with constant literal temporaries on the RHS
- Instructions with calls to Slate inverses and solves [2]

Algorithm 2: PyOP2 vectorisation

```
1  if PyOP2 kernel is vectorisable then
2      Change target of the wrapper kernel to CVectorExtensionsTarget;
3      Inline all Slate and TSFC kernels in the PyOP2 kernel;
4      Align all temporaries;
5      forall vectorisable instructions do
6          Shift the bound of the loop ( iname ) to vectorise over so that it starts from 0;
7          Split the iname according to the SIMD length of the architecture;
8          Add a new axis to the temporaries and index it with the provided iname;
9          Tag axes to vectorise over;
10         Tag iname to vectorise over with;
```

3.3.2 An example

Consider an arbitrary linear variational form, three Slate tensors T_1, T_2, T_3 defined on it and an assembly of $T_0 + T_1 - T_2$. In the following it will be explained step-by-step how the kernel changes when it is vectorised. To facilitate comparisons, both vectorised and unvectorised kernels are shown with all subkernels inlined. In reality, the subkernels of the unvectorised kernel are typically not inlined.

Both vectorised and unvectorised kernels are defined through a kernel name and some arguments, in particular some coefficient vectors whose names start with `dat`, and loop bound parameters `n_start` and `n_end`.

The loops (inames) are defined through domains and tags. The outer loop here is called `n`. In the unvectorised kernel, the loop has bounds `start` and `end` and is untagged. In the vectorised kernel, the loop is split into `n_shift_outer` and `n_shift_batch`. The iname `n_shift_outer` starts from 0 and only runs to a fourth of the original loop length, and `n_shift_batch` has a loop length of 4. This transformation is realised in line 6, 7, 10 in algorithm 2.

[1]This is unproblematic because the assembly time is typically not critical for matrix-free solvers which, by definition, do not assemble matrices. Where matrices are assembled, we usually expect solve time to dominate.

[2]Gcc could do this with an implementation of solve and inverse callables in PyOP2 in a strided fashion, but this is not possible with clang due to a missing support of addressing variables of vectorextended type

```
DOMAINS: /*unvectorised*/
[n_end, n_start] -> { [n] : n_start <= n < n_end }
{ [i_id_8] : 0 <= i_id_8 <= 5 }
{ [i_1] : 0 <= i_1 <= 5 }
...
--------------------------------------------------------------------------

INAME TAGS: /*unvectorised*/
n: None
i_1: None
i_id_8: None
...
```

```
DOMAINS: /*vectorised*/
[n_end, n_start] -> {[n_shift_outer, n_shift_batch] : n_shift_batch >= 0
                      and -4n_shift_outer <= n_shift_batch <= 3
                      and n_shift_batch < n_end - n_start - 4
   n_shift_outer }
{ [i_id] : 0 <= i_id <= 5 }
{ [i] : 0 <= i <= 5 }
...
--------------------------------------------------------------------------

INAME TAGS: /*vectorised*/
n_shift_batch: vec
n_shift_outer: None
...
```

Figure 3.7: Domains and iname tags of an unvectorised and a vectorised Loo.py kernel for $T_0 + T_1 - T_2$

Further, the temporaries are broken by adding another axis and indexing into it with the iname that has the vec tag, see the `dim_tags`. This is implemented in line 8-9 in algorithm 2.

```
TEMPORARIES: /*unvectorised*/
T0: type: np:dtype('float64'), shape: (6),
    dim_tags: (N0:stride:1) aspace:local
t0: type: np:dtype('float64'), shape: (1, 3, 2),
    dim_tags: (N2:stride:6, N1:stride:2, N0:stride:1) aspace:auto
...
```

```
TEMPORARIES: /*vectorised*/
T0: type: np:dtype('float64'), shape: (6, 4),
    dim_tags: (N0:stride:4, vec) aspace:local
t0: type: np:dtype('float64'), shape: (1, 3, 2, 4),
    dim_tags: (N2:stride:24, N1:stride:8, N0:stride:4, vec) aspace:auto
...
```

Figure 3.8: Temporaries of an unvectorised and vectorised Loo.py kernel for $T_0 + T_1 - T_2$

Based on the previous transformations the instructions between the unvectorised and vectorised kernels differ as follows.

```
INSTRUCTIONS:/*unvectorised*/
for n
    ...
    for i_id_8
        T0[i_id_8] = 0.0                                        {id=init0}
    end i_id_8
    ...
    for i_1
        t0[0, i_1 // 2, i_1 + (-2)*(i_1 // 2)] = \
            t0[0, i_1 // 2, i_1 + (-2)*(i_1 // 2)] \
            + (-1.0) * T2[i_1] + T0[i_1] + T1[i_1]              {id=insn_4}
    ...
end n
```

```
INSTRUCTIONS: /*vectorised*/
for n_shift_batch, n_shift_outer
    ...
    for i_id
        T0[i_id, n_shift_batch] = 0.0                           {id=init0}
    end i_id
    ...
    for i
        t0[0, i // 2, i + (-2)*(i // 2), n_shift_batch] = \
            t0[0, i // 2, i + (-2)*(i // 2), n_shift_batch] \
            + (-1.0)*T2[i, n_shift_batch] \
            + T0[i, n_shift_batch] \
            + T1[i, n_shift_batch]                              {id=insn_4}
    end i
    ...
end n_shift_batch, n_shift_outer
```

Figure 3.9: An unvectorised and vectorised C kernel for $T_0 + T_1 - T_2$

Loo.py further translates these kernels into C kernels, see figure 3.10. The variables in the vectorised kernels are extended with `__attribute__ (( vector_size (32) ) )` and aligned with `(( aligned (64) ) )`. The loops are split equivalently to the previously presented Loo.py kernel. The linear algebra operations are executed on four tensors at a time in the bulk slab. The vectorised local assembly is not presented in this listing to keep the code accessible to the reader, but examples can be found in [28]. The final slab contains the unvectorised instructions for the remainder of the loop.

```
1
2  void wrap_slate_wrapper(int32_t const start, int32_t const end,
3                          double *__restrict__ dat0,
4                          double const *__restrict__ dat1,
5                          double const *__restrict__ dat2,
6                          double const *__restrict__ dat3,
7                          double const *__restrict__ glob0,
8                          double const *__restrict__ glob1,
9                          int32_t const *__restrict__ map0)
10 {
11   double __attribute__((vector_size(32))) T0[6] \
12          __attribute__ ((aligned (64)));
13   double __attribute__((vector_size(32))) T1[6] \
14          __attribute__ ((aligned (64)));
15   double __attribute__((vector_size(32))) T2[6] \
16          __attribute__ ((aligned (64)));
17   double __attribute__((vector_size(32))) t0[3 * 2] \
18          __attribute__ ((aligned (64)));
19
20   /* ... more initialisations ... */
21
22   /* bulk slab */
23   for (int32_t n_shift_outer = 0;
24        n_shift_outer <= -2 + -1 * start + (3 + end + 3 * start) / 4;
25        ++n_shift_outer)
26   {
27
28     /* ... filling T* with data through vectorised local assembly ... */
29
30     for (int32_t i = 0; i <= 5; ++i)
31       t0[i] = t0[i] + -1.0 * T2[i] + T0[i] + T1[i];
32   }
33
34   /* ... final slab ... */
35 }
```

Figure 3.10: A vectorised C kernel for $T_0 + T_1 - T_2$

3.4 Results

In order to simplify a comparison of the already existing vectorisation of operator actions in Firedrake, the results in [28] were reproduced first, see section 3.4.3, and the experiments were extended to local linear algebra kernels on the same operators as in [28] after that. A reproduction of the results in [28] is useful, because the experiments are run on a different architecture, with a newer compiler version and newer components of Firedrake.

The main difference of the results in section 3.4.3 to the results in [28] is that they were produced on an architecture which has dual sockets and the compiler version are gcc-11.2 and clang-13.01. In the later compiler versions auto-vectorisation has been improved so a lower performance for the vectorised kernels relative to the baseline is expected. Because the hardware used has different specifications to [28] the setup has been adapted so that the whole machine is occupied in the experiments, see section C.1 in the appendix.

The code for the experimental evaluation (data collection and visualisation) has been adapted from [28]. The baseline run is an assembly without explicit vectorisation but with auto-vectorisation. The high performance run is explicitly vectorised with a cross-element vectorisation strategy.

3.4.1 Software and hardware details

The Firedrake software version used for the following experiments is published under [72]. The PETSc software was built separately. It is published under [73]. The Zenodo directory for the experiments including data, plots and scripts can be found in [74].

	Intel(R) Xeon(R) CPU E5-2640 v3
Base Frequency	2.6
Sockets	2
Cores per socket	8
Threads per core	2
SIMD instruction set	AVX2
FMA units per core	2
Theoretical peak performance (double-precision)	665.6 GFLOP/s [3]
Memory bandwidth performance	60 GB/s [4]
GCC compiler version	11.2
GCC compiler flags	-fPIC -Wall -std=gnu11 -march=native -O3 -ffast-math -fopenmp
Clang compiler version	13.01
Clang compiler flags	-fPIC -Wall -std=gnu11 -march=native -O3 -ffast-math -fopenmp-simd

Table 3.4: Hardware specification for the dual-socket Intel(R) Xeon(R) CPU E5-2640 v3 architecture

3.4.2 Problem setup

Performance measurements were taken for the assembly of a variety of FE operator actions. Let the operators involved (mass, Helmholtz, Laplacian, elasticity and hyperelasticity operators) be denoted by a. The formulation of the variational forms for these operators is given in the supplemental material of [28]. The complexity of the operators is correlated to their arithmetic intensity, see figure 3.12. The problems are discretised with continuous Lagrange elements (CG) on a triangular, quadrilateral, tetrahedral and hexahedral mesh. The operators are applied to a coefficient vector f with $l = \texttt{action}(a, f)$. First, results for assembling l are presented. These results are a reproduction of what was shown in [28] on a different architecture.

Further, performance results for the vectorised assembly of Slate expression were produced. The Slate expression $l = \texttt{Tensor}(\texttt{action}(a, f)) + \texttt{Tensor}(\texttt{action}(a, f)) - \texttt{Tensor}(\texttt{action}(a, f))$ is assembled. Note that nothing cancels out in the expression. The performance results for the vectorisation of the corresponding Slate kernels are a proof of concept. A note on the extension to a more realistic setting can be found in section 5.6.

3.4.3 Results for an operator action and a Slate expression on operator actions

The results of [28] were successfully reproduced accounting for the fact that they were produced on the same micro-architecture but in a dual-socket setup. Therefore, the theoretical peak and memory bandwidth performance of the machine is a double of the one in [28]. The baseline performance is capped at around 200 GFLOPS/s which is 30 percent of the theoretical peak performance, whereas the explicitly vectorised performance is capped at around 400 GFLOPS/s which is 60 percent of the theoretical peak performance. Those numbers match the study in [28].
The majority of measurement points is further from the peak bandwidth-dominated limit than in the previous study, however. A reason for that could be that the measured bandwidth in [28] was only probed with 2 threads. That is typically insufficient to get a realistic limit for the memory bandwidth, for further explanations see the section B in the appendix.

Comparing the achieved performance for the assembly of a Slate expression involving linear algebra operations on three operator actions to a single operator action assembly, the majority of cases achieve a similar FLOPS processing rate. The arithmetic intensity of the kernels is about three times higher for the Slate expression compared to the operator action which matches the expectations.

The performance for the quadrilateral and hexahedral test cases seems poor compared to the simplicial case. The reason for that is that the latter cases are not sumfactorised and have therefore a much higher arithmetic intensity.

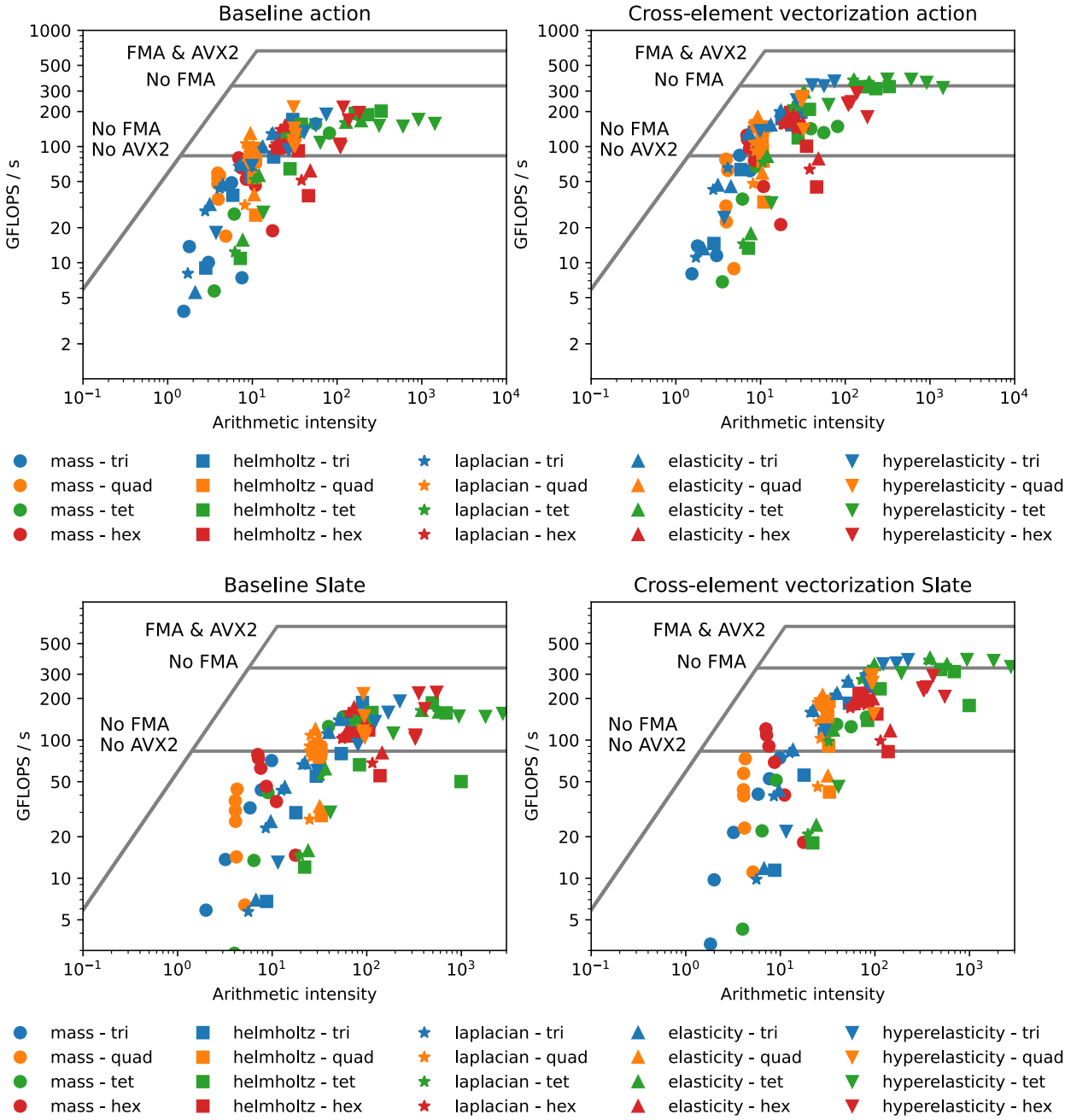


Figure 3.11: Roofline plot for a variety of variational forms, polynomial degrees and meshes for the assembly of an operator action (top) and Slate expression of operator actions (bottom). While the top plots are a reproduction of [28] for the vectorisation of TSFC kernels on a new machine, the bottom plots present new results for the vectorisation of Slate kernels. The bandwidth limit is measured with a benchmark, the FLOP limits are theoretical.

The speedup achieved by explicit vectorisation over auto-vectorisation for each test case can be found in figure 3.12. In the worst cases there is no speedup but the performance is equivalent to the auto-vectorised kernels. Again, it can be seen that the speedup increases with the complexity of the kernels.

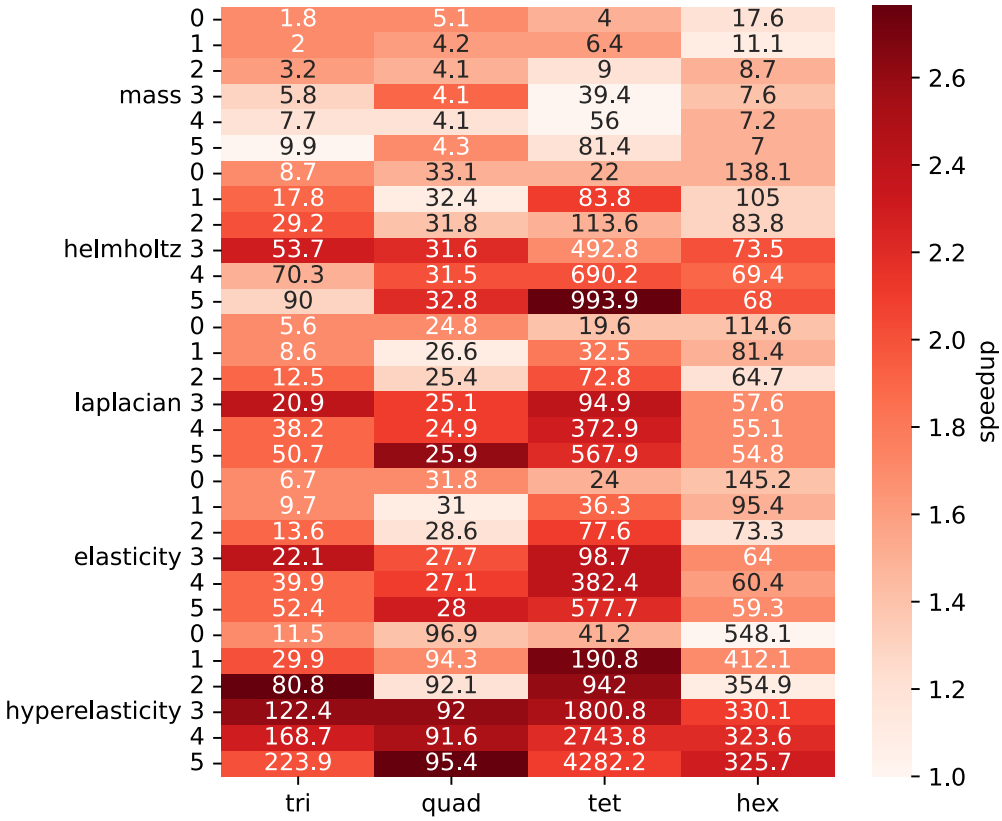


Figure 3.12: Speedup of explicit vectorisation over auto-vectorisation for the assembly of a Slate expression involving three operator actions. The arithmetic intensity is specified in the annotations on top of the colored boxes for the speedup.

The throughput plots in figure 3.13 and 3.14 show that the performance gain through vectorisation is higher for test cases with a high arithmetic intensity, which is expected because the CPU spends more time on the calculations rather than waiting for the data to be loaded, as already shown in [28]. Explicit vectorisation has higher performance for almost all test cases besides the ones where the operator is not complex enough, in particular the mass operator and low order polynomial degrees.

Tables with arithmetic intensities, speedup numbers and other metrics for both the assembly of operator actions and a Slate expression of operator actions are appended in table C.3 and C.2.

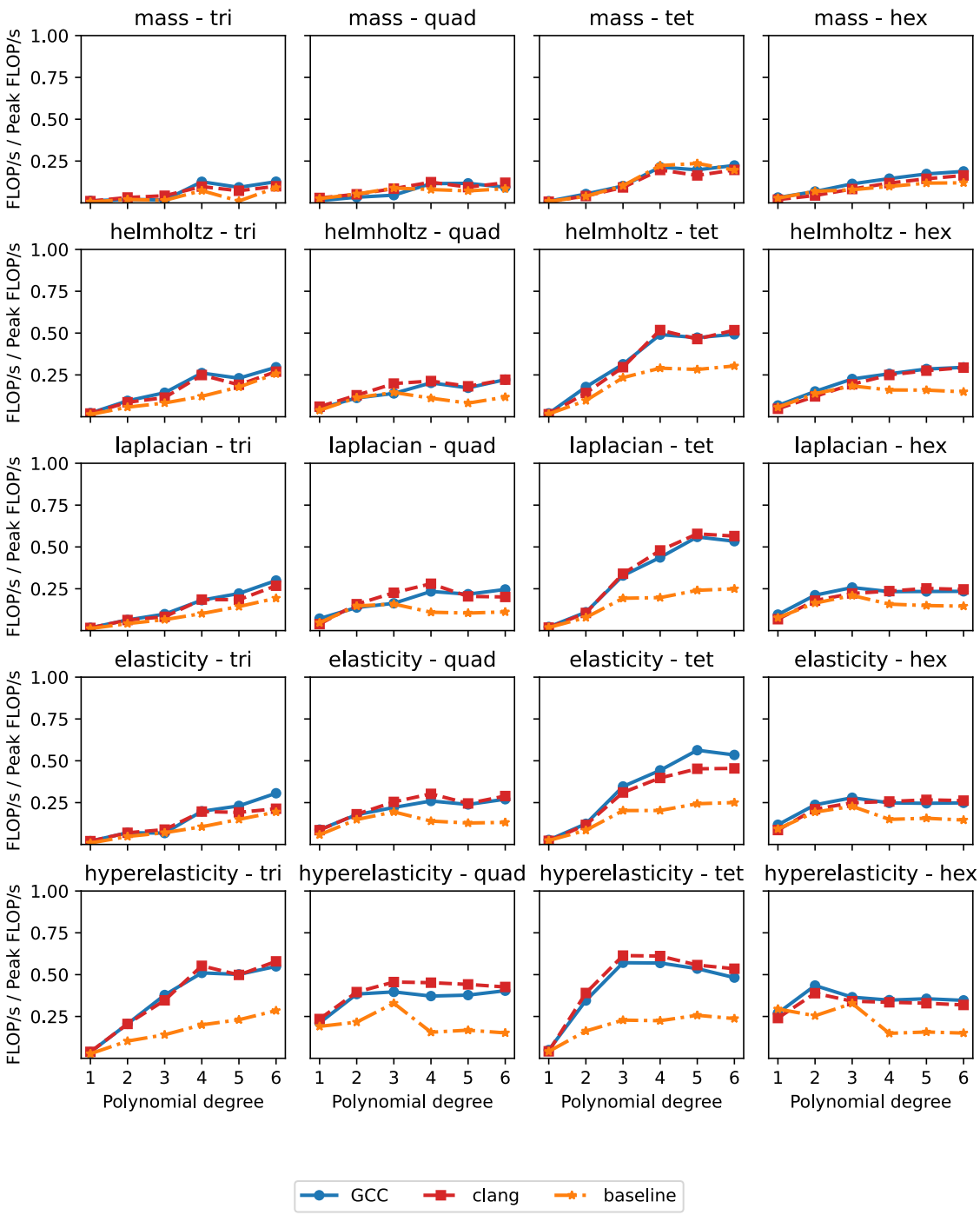


Figure 3.13: Throughput [FLOP/s] scaled by peak throughput of the test hardware for an operator action on a variety of variational forms, polynomial degrees and meshes.

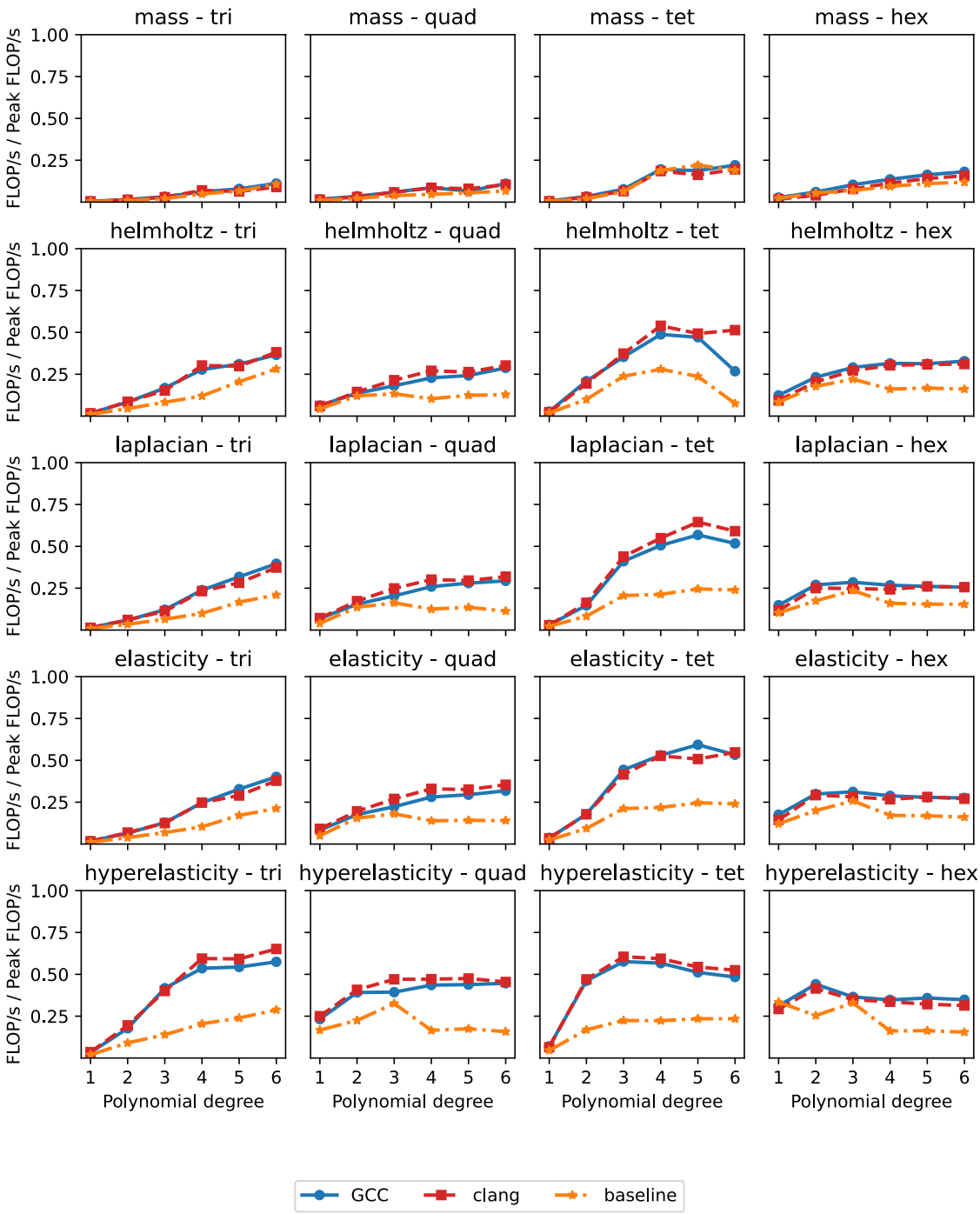


Figure 3.14: Throughput [FLOP/s] scaled by peak throughput of the test hardware for a Slate expression on a variety of variational forms, polynomial degrees and meshes.

3.5 A summary of the contributions

In this chapter existing language and compiler infrastructure in Firedrake, and my extensions to them for the code generation of local linear algebra expression are presented.

The drawbacks of Slate's previous compiler are outlined, in particular a lack of extensibility to generate vectorised kernels and the missed opportunity for reuse of already existing performance optimisation infrastructure in Firedrake's codebase.

I have changed the compiler from translating Slate to an Eigen and C++ backend to translating Slate to a Loo.py backend, because Loo.py presents an easy pathway for vectorisation due to its ability to manipulate loop structures. The compilation from Slate to Loo.py has been implemented with an intermediate stage in GEM so that performance optimisation from TSFC can be recycled. Most of the Slate operations can be straightforwardly translated into GEM, but some new nodes in Slate and GEM were needed for the more complex linear operations (inverse and solve) as well as an accompanying translation to Loo.py.

Further, I demonstrated that the Slate kernel vectorisation is functional, achieves a high amount of peak performance for a simple linear algebra expression on a variety of kernels and polynomial degrees.

Chapter 4

An automated framework for matrix-free local linear algebra

Solving big equation systems, if they arise from high order FE discretisations or other numerical methods, poses two problems. One is the high storage requirement and the load bottlenecks of the global system matrices, the second is the number of operations it takes solve the system. Both problems are an immediate consequence of the size of the matrices involved. Matrix-free methods can be used to avoid building the matrices explicitly. As mentioned in section 2.3.1 the core idea of matrix-free methods is to solve the equation system deduced from FE problems with Krylov subspace methods. Krylov subspace methods only involve matrix-vector products, and in matrix-free methods, the matrix-vector multiplications are replaced by a parametrised assembly of the operators.

Globally matrix-free methods were introduced in Firedrake as a byproduct of the work presented in [75]. This is sufficient for all solvers, except those which require linear algebra operations on the local assembly tensors. In that case, globally matrix-free methods assemble the local matrices but apply the global operator matrix-free. In particular using high-order discretisations, these local tensors obtain such a large size that they introduce a store and load bottleneck for preconditioners such as the hybridisation preconditioner, see section 2.2.3, or Gopalakrishnan-Tan multigrid, see section 2.2.4.

In this chapter, an application is presented for which fully matrix-free methods involving local linear algebra operations is required. This application motivates the necessity of the new code infrastructure presented after that. The new infrastructure I put into place in Firedrake allows an automatic code generation for fully matrix-free preconditioners involving local linear algebra expressions. The locally matrix-free solvers can be easily controlled through parameters mimicking PETSc options. While this is driven by compatible FE discretisations and their hybridisation in this thesis, it is not limited to that. For example, the local matrix-free infrastructure could be used for CG FE methods where facet and intra-elemental contributions are separated in the blocks of a mixed matrix and the systems are solved through static condensation.

4.1 A model problem: a hybridised, mixed Poisson problem

One possible application of fully matrix-free hybridisation is a Poisson problem. Poisson problems need to be solved in various physical setups. One of them is the Navier-Stokes equations: If the time derivative in the incompressible Navier-Stokes equations is solved with a predictor-evaluation-corrector scheme, see e.g. [76] , a Poisson problem has to be solved at every time step for updating the pressure with the velocity prediction. The Poisson problem for the pressure can be rewritten into a mixed system by introduction of a flux variable. If the mixed spaces are chosen in a compatible way this has advantages, e.g. the conservation of some physical quantities, refer to section 2.1.2.2.1. Mixed Poisson problems are difficult to solve because a) they have saddle-point structure and are indefinite and b) they are globally coupled and the condition number of the operator grows when refining its approximation. Hybridisation is used to remove most of the global coupling.

The hybridisation of a mixed Poisson problem was introduced in section 2.2.3, where hybridisation is considered in a *locally matrix-explicit* form. Now, in section 4.1.2, it is outlined which mathematical changes are required to set up hybridisation to work well using *locally matrix-free* solvers. In particular in section 4.1.2.2, the setup and results of a numerical investigation to find out which local solvers work well for the mixed Poisson problem are presented.

4.1.1 The bottleneck of locally matrix-explicit hybridisation

In order to demonstrate that hybridisation with *locally matrix-explicit* methods is insufficient when discretisations with a high polynomial approximation degree are considered, the hybridisation example in 2.2.1 and 2.2.2 is revisited. The focus is set on solving the hybridised mixed Poisson problem on a hexahedral mesh with *globally matrix-free methods* for now.

In *fully matrix-explicit* hybridisation the expression that has to be globally assembled is a Slate expression for the Schur complement on each cell. Solving with *globally matrix-free* hybridisation means that this matrix in the global equation system is never built explicitly, instead its actions on vectors are used to move the solution forward in each iteration. In globally, but not locally, matrix free methods a vector is assembled per cell and distributed onto the global level, but the vector per cell is assembled as a multiplication of the local Schur complement with a local coefficient vector, and the local Schur complement is still built in a matrix-explicit way.

Here, the solver for the global trace system is a Jacobi preconditioned CG method. For the mixed Poisson problem the solver is not robust in polynomial order of the discretisation. This means that the finer the discretisation is per cell, the more iterations are needed to get a global solution.

Therefore, the complexity of the trace solve grows with the polynomial order not only due to the increasing complexity of the local kernel, but also due to the increasing amount of iterations of the global solver. The missing robustness with approximation degree can be remedied by choosing a different solver, e.g a Gopalakrishnan-Tan multigrid see section 2.2.4, results for which are presented in chapter 5. The fact that the iteration count increases with polynomial order in this example, should be taken into account evaluating the following performance profiles.

Performance profiles of the hybridisation for the mixed Poisson problem solved with a globally, but not locally matrix-free method for a low order discretisation and a higher order one are shown in figure 4.1.

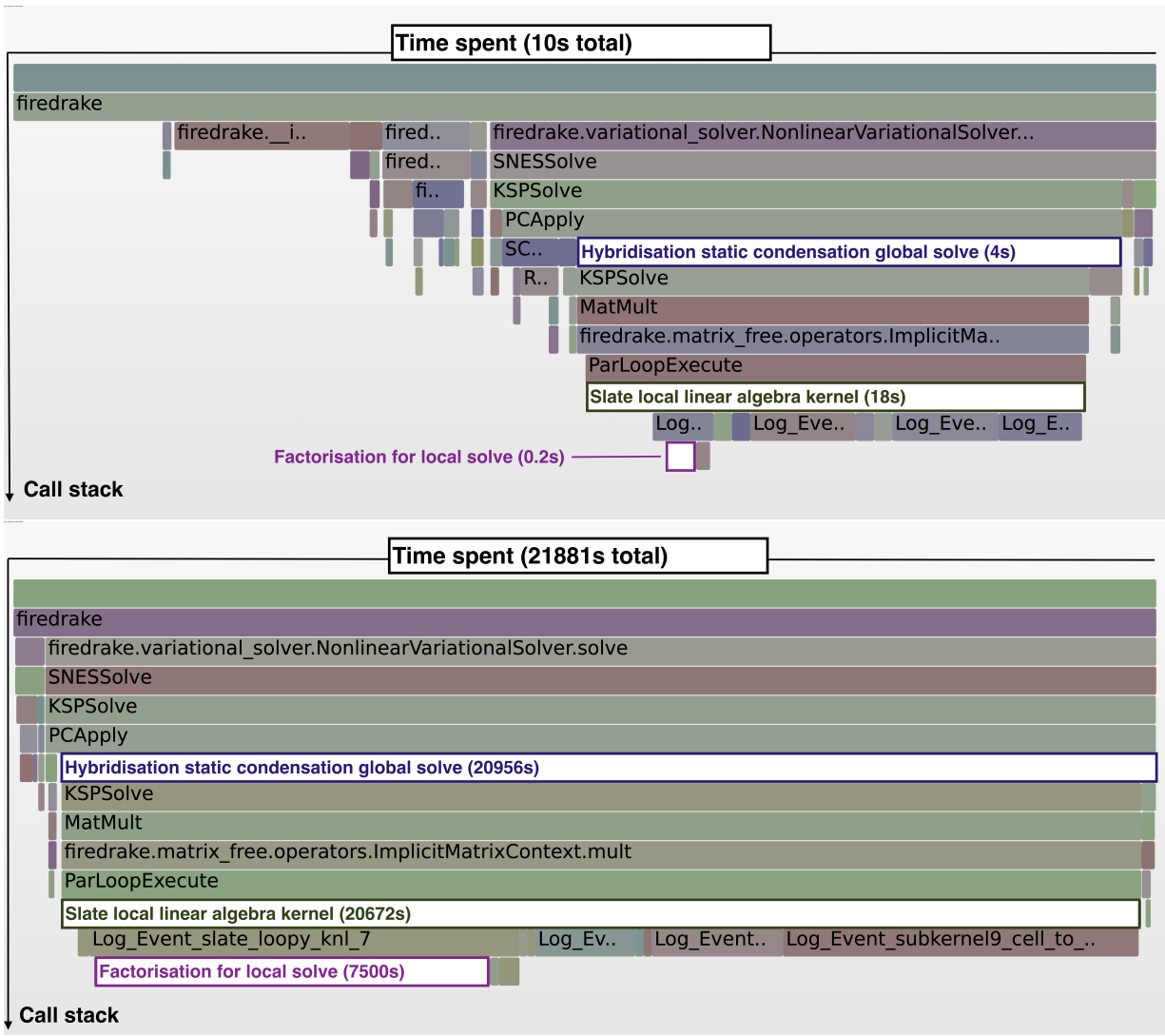


Figure 4.1: Performance profile of a mixed Poisson problem discretised with **RT1-DG0** (top) and **RT5-DG4** (bottom) solved with hybridisation and a Jacobi preconditioned conjugate gradient method on the trace solve utilising global but not local matrix-free infrastructure. The local factorisation dominates the solve time for high order.

In the top of figure 4.1, in the profile for the low order discretisation, the local solve is part of the last row of measurements and so small it did not even get any text assigned in the flamegraph. Comparing that to the profile for the high order discretisation, a dominance of the local solve kernels can be seen, in particular the factorisation denoted by `solve_getrf` as part of the static condensation solve `SCSolve`. This bottleneck is mostly caused by increasing sizes of the local tensor and it is the reason why it is critical to use fully matrix-free hybridisation on high order discretisations. While calculating and storing the factorisation for the solve upfront might improve the runtime, it is not feasible to store a matrix coming from a high order discretisation for every single cell in the mesh.

The bottleneck is also partially caused by the increasing amount of iterations of the global solver. In order to prove that not the increase in global solvers iterations is the dominating reason for the dominance of trace solve in the performance profile, consider the following plots. Comparing the local and global solve times in figure 4.2 on the left and their ratio on the right, normalised per iterations, one can see the that the local solve makes up more of the trace solve with an increasing approximation degree.

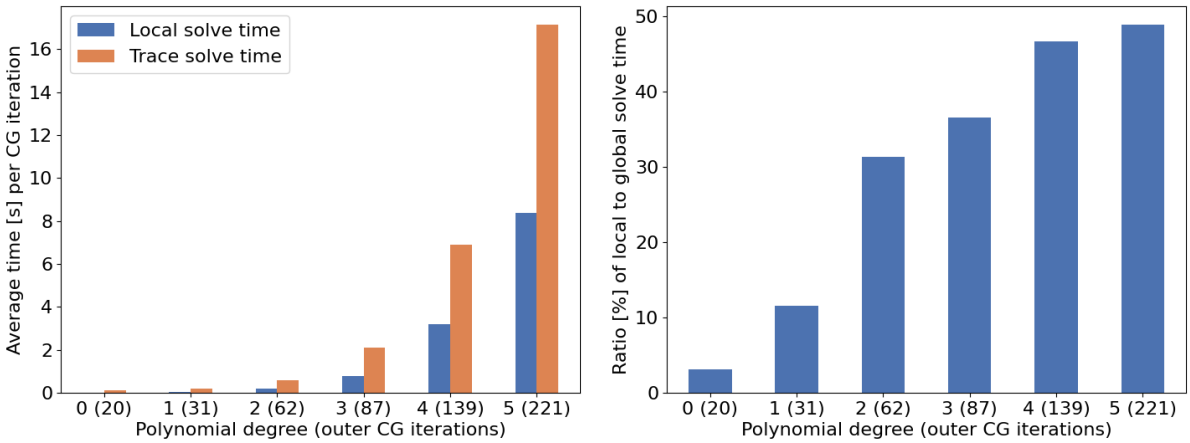


Figure 4.2: Average time of local and global solve per iteration (left) and percentage of time spent in local versus global solve iteration (right) for the global, static condensation solve in the hybridisation of a mixed Poisson problem on a hexahedral mesh.

4.1.2 The numerics of fully matrix-free hybridisation

In order to solve the systems involved in the hybridisation of a mixed Poisson problem in a *fully matrix-free* manner, some of the solving steps have to be altered as explained in section 4.1.2.3, in particular the steps for the static condensation in example 2.2.2. Further, the numerical properties of the local matrices involved in the hybridisation of the mixed Poisson problem are explained in section 4.1.2.1.

4.1.2.1 Numerical properties of the matrices involved

Saddle point systems can be solved with Schur complement reduction techniques. A block matrix $\mathscr{A}$ with 2×2 blocks allows for a Schur decomposition as long as its A_{00}-block is not singular. Then, the original block matrix is only singular if and only if the Schur complement is singular [18].

Applying that to the hybridised mixed Poisson problem, consider that the operator of the hybridised mixed Poisson problem consists of 3×3 blocks for interactions of intra-cell velocity, pressure and inter-cell velocity, refer to section 2.2.3 for more details.

$$\mathscr{A} = \begin{pmatrix} \mathbf{A}_{00} & \mathbf{A}_{01} & \mathbf{A}_{02} \\ \mathbf{A}_{10} & \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{20} & \mathbf{A}_{21} & \mathbf{A}_{22} \end{pmatrix} \tag{4.1}$$

The system is a saddle-point system, there are no interactions between the velocities on different element faces, A_{22} is a zero block. The block-structure of a hybridised system operator is visualised in figure 4.12.

It is possible to employ a Schur reduction technique since the block $\begin{pmatrix} \mathbf{A}_{00}\mathbf{A}_{01} \\ \mathbf{A}_{10}\mathbf{A}_{11} \end{pmatrix}$ corresponding to intra-cell velocity and pressure is not singular for the hybridised mixed Poisson problem. In the static condensation step for the hybridised system, intra-cell velocity and pressure are eliminated, a Schur complement $S_\lambda \Lambda$ is solved globally for the velocity on the traces, see equation (4.2), and then intra-cell velocity and pressure are reconstructed, see equations (4.5) and (4.6).

Global trace solve:

$$\mathbf{S}_\lambda \Lambda = \mathbf{E} \tag{4.2}$$

$$\text{with} \qquad \mathbf{S}_\lambda^K = \mathbf{A}_{22}^K - \left[\mathbf{A}_{20}^K, \mathbf{A}_{21}^K\right] \begin{bmatrix} \mathbf{A}_{00}^K & \mathbf{A}_{01}^K \\ \mathbf{A}_{10}^K & \mathbf{A}_{11}^K \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{A}_{02}^K \\ \mathbf{A}_{12}^K \end{bmatrix} \tag{4.3}$$

$$\text{and} \qquad \mathbf{E}^K = \mathbf{F}_2^K - \left[\mathbf{A}_{20}^K, \mathbf{A}_{21}^K\right] \begin{bmatrix} \mathbf{A}_{00}^K & \mathbf{A}_{01}^K \\ \mathbf{A}_{10}^K & \mathbf{A}_{11}^K \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{F}_0^K \\ \mathbf{F}_1^K \end{bmatrix} \tag{4.4}$$

Local reconstruction calls:

$$\mathbf{P}^K = \mathbf{S}_{11}^K.\text{solve}\left(\mathbf{F}_1^K - \mathbf{A}_{10}^K \left(\mathbf{A}_{00}^K\right)^{-1} \mathbf{F}_0^K - \mathbf{S}_{12}\Lambda^K\right) \tag{4.5}$$

$$\mathbf{U}^K = \mathbf{A}_{00}^K.\text{solve}\left(\mathbf{F}_0^K - \mathbf{A}_{01}^K \mathbf{P}^K - \mathbf{A}_{02}^K \Lambda^K\right) \tag{4.6}$$

$$\text{with} \qquad \mathbf{S}_{1i}^K = \left(\mathbf{A}_{1i}^K - \mathbf{A}_{10}^K \left(\mathbf{A}_{00}^K\right)^{-1} \mathbf{A}_{0i}^K\right) \tag{4.7}$$

Note, that the matrices in equation (4.3) to (4.6) are local matrices. There are three matrices which need to be locally inverted: the mixed matrix $\begin{pmatrix} \mathbf{A}_{00} \mathbf{A}_{01} \\ \mathbf{A}_{10} \mathbf{A}_{11} \end{pmatrix}$, the intra-cell velocity mass matrix $\mathbf{A}_{00}$ and the Schur complement with respect to the pressure $\mathbf{S}_{11}$. For high order FEM that is problematic, because even the local matrices are big. Replacing the inverse with an iterative solve can yield a faster a solution, for more details see the later chapter 4.2.1.2.

The mixed matrix is locally like a mixed Poisson problem. There are no interactions of pressure with pressure in the cell, it has saddle-point structure, and is indefinite. A_{00} on the other hand is positive-definite and symmetric, and so is the inner Schur complement. While A_{00} is diagonal dominant, the inner Schur complement is not. All numerical properties presented in this section have been examined experimentally with a script which can be found in [77] in the investigation folder.

4.1.2.2 A local solver for the velocity mass matrix and the pressure Schur complement

In the local reconstruction calls in equation (4.5) and (4.6) there are two local solves, one involving the velocity mass matrix and the other one involving the pressure Schur complement. In order to solve the systems fully matrix-free, these local solves are replaced by local, matrix-free solves. In general a good, iterative solver would converge in a number of iterations which is independent of the approximation parameters (mesh parameters and approximation degree) and with less work than a direct solve. Both the velocity mass matrix $\mathbf{A}_{00}$ and the Schur complement $\mathbf{S}_p^K (= \mathbf{S}_{11})$ have a growing condition number with an increasing degree of approximation polynomial and therefore, a robust solver requires some extra work. I performed a numerical investigation to find good local preconditioners for this problem in order to keep the condition number of the operators in the local solves low.

In the investigation a mixed Poisson problem is solved on a single cell, hexahedral mesh with help of Firedrake and PETSc. In order to construct PETSc solvers on the velocity mass matrix and the pressure Schur complement, PETSc's Schur decomposition fieldsplit preconditioner is used. Explanations on the preconditioner can be found in [78]. The notion of *fieldsplit* solver denotes the solver to invert the matrices needed in the inverse of the Schur decomposition. The solves on the fieldsplits are iterated to an accuracy that is sufficient for the outer solve around the Schur decomposition fieldsplit preconditioner to converge in one iteration.

The best performing Schur fieldsplit preconditioner from the investigation uses Conjugate Gradient on both fieldsplits, with a diagonal preconditioner for the velocity mass matrix solve and a diagonal of the interior penalty discontinuous Galerkin formulation of the Laplacian on the pressure Schur complement solve.

The reasoning for the preconditioner on the Schur complement solve is that the Schur complement is spectrally equivalent to a Laplacian in the DG space. A code example for this operator can be found in the details of the local solves later in figure 4.19. The investigated fieldsplit solvers are also used to invert the mixed matrix in the trace solve as is explained in section 4.1.2.3.

The solver iterations are investigated for three types of cells: a cell which is scaled by a scaling factor (i.e. a cube but not a unit cube), an affinely deformed cell (a cuboid) and a non-affinely deformed cell where two neighbouring corners are moved, one by a deformation factor and the other by 75 percent of that deformation factor. The solve iterations are calculated by the following, where fsp1 denotes the pressure Schur complement solver and fsp0 the velocity mass matrix solver.

$$\text{total iterations} := \text{its}_{\text{outer}} \cdot \left(\text{its}_{\text{fsp1}} + \text{its}_{\text{fsp1}} \cdot \text{its}_{\text{fsp0}}\right) \tag{4.8}$$

The value is an approximation because the iteration count of the velocity mass matrix is solved once per pressure Schur complement solver iteration and the iterations count for fsp0 is the one of the last fsp1 iteration.

The iteration counts for the undeformed, unit cell are presented in table 4.1. The velocity mass matrix solver is robust across approximation degrees and the pressure Schur complement solver is not. It converges in a reasonable range of iterations, however, involving less work than a direct solve would need.

Discretisation	RTCF1-DG0	RTCF2-DG1	RTCF3-DG2	RTCF4-DG3	RTCF5-DG4	RTCF6-DG5
Total iteration count	2	2	8	8	20	18
fsp0 iteration count	1	1	1	1	1	1
fsp1 iteration count	1	1	4	4	10	9

Table 4.1: Iteration counts on an undeformed, unit cell. The final row shows that solver of the pressure Schur complement is not p-robust.

Further, the first set of figures 4.3, 4.4 and 4.5 present the amount of total solver iterations for a range of deformation factors for the three types of cells, and the condition number of the mixed Poisson operator on those deformed cells. The outer solver iterations are all 1 and the corresponding plots can be found in the appendix D. The next set of figures from 4.6 to 4.11 show the iteration counts and condition numbers per fieldsplit solver for each type of cell. In the following, both sets of figures, showing total and fieldsplit solver iteration counts, are examined in conjunction.

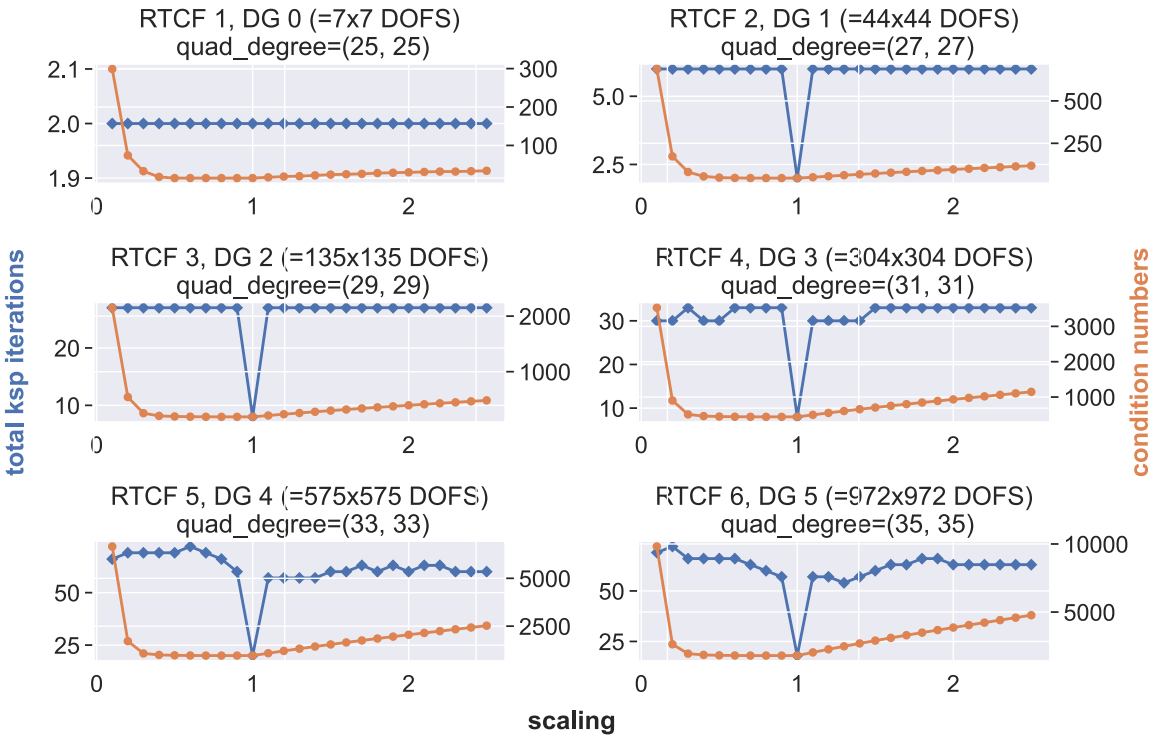


Figure 4.3: Scaled cells: Total solver iterations per scaling factor

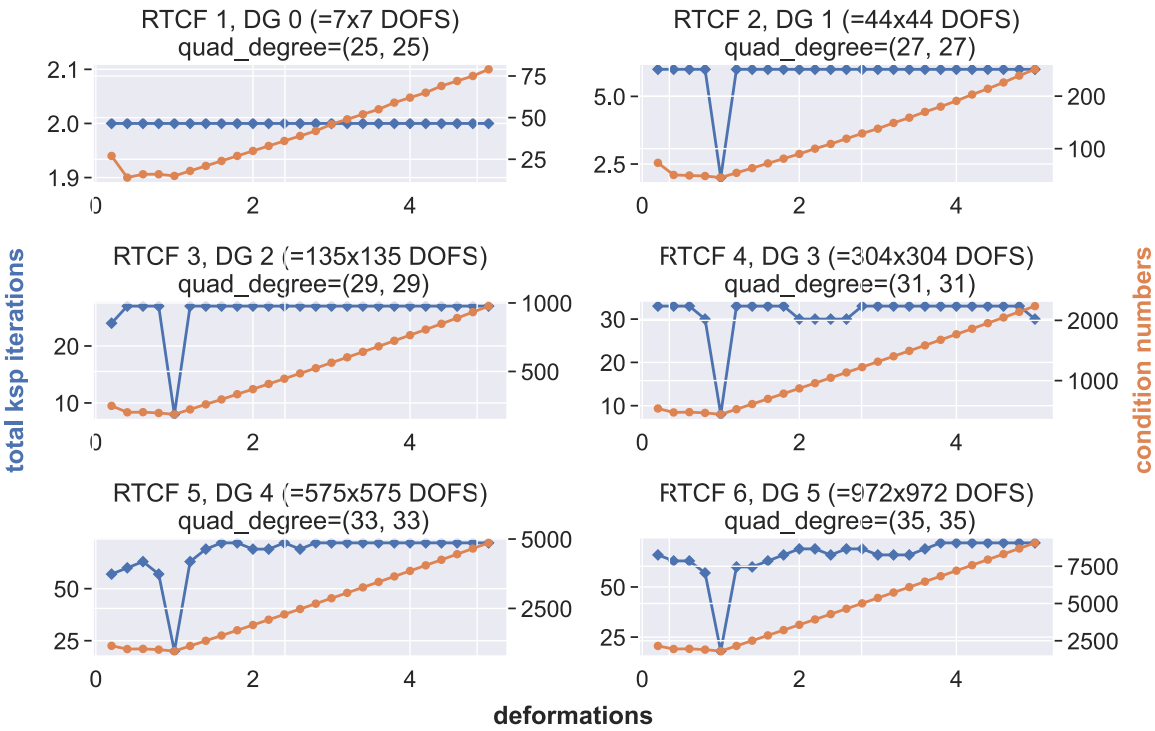


Figure 4.4: Affinely deformed cells: Total solver iterations per deformation factor

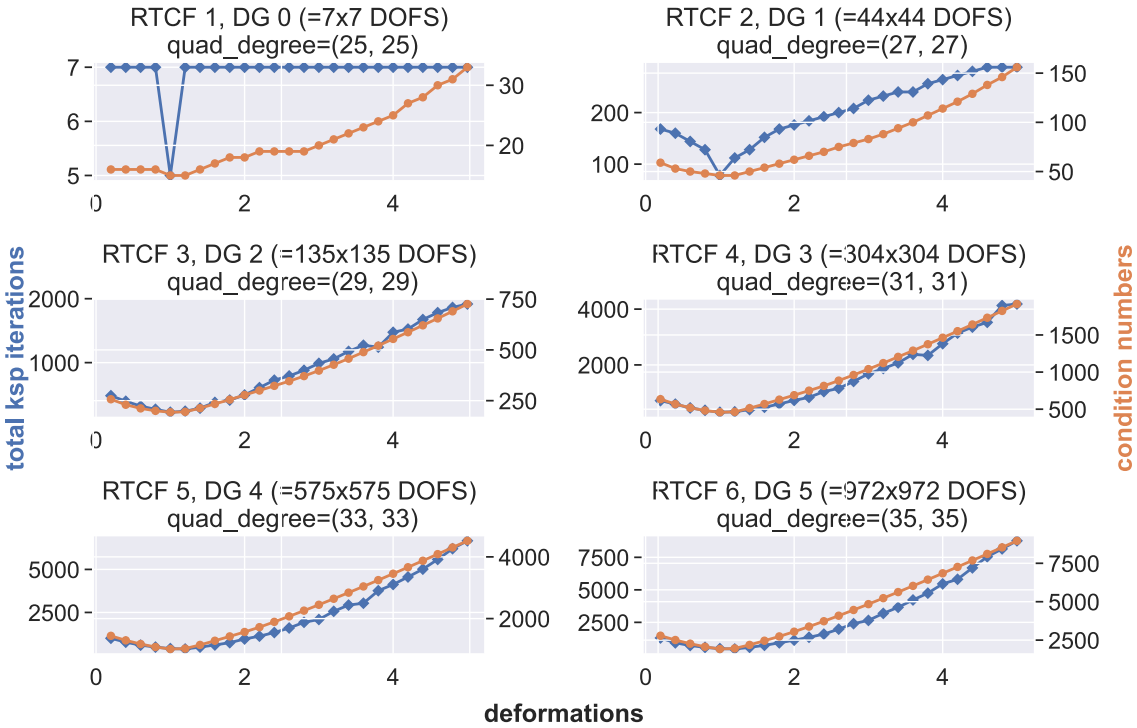


Figure 4.5: Non-affinely deformed cells: Total iterations per deformation factor

The total iteration count is approximately robust across deformation factors for scaled and affinely deformed cells, see figure 4.3 and 4.4. Further, the velocity mass matrix solver has a fixed iteration count across approximation degrees in these cases, see figure 4.6 and 4.8. The iteration count of the other fieldsplit solver, the one for the pressure Schur complement, grows with the approximation degree, see figure 4.7 and 4.9 but stays in a reasonable range involving less work than a direct solve would need.

The total iteration count is not robust on non-affinely deformed cells for any higher than lowest order discretisation and grows with the condition number, as can be seen in figure 4.5. On this cell type the total iteration count is growing with the deformation factor due to an increasing number of iterations on both fieldsplit solvers with the deformation factor, see figure 4.10 and 4.11.

Given the results from the numerical investigation here, new infrastructure has been introduced in the Slate based preconditioners in Firedrake to enable a) locally matrix-free matrix-vector products and solves see section 4.2.2 and 4.2.3 b) user-defined local preconditioners to support e.g. the DG Laplacian, see section 4.2.4.1, and c) local preconditioners built with the diagonal of a local matrix, see section 4.2.4.2.

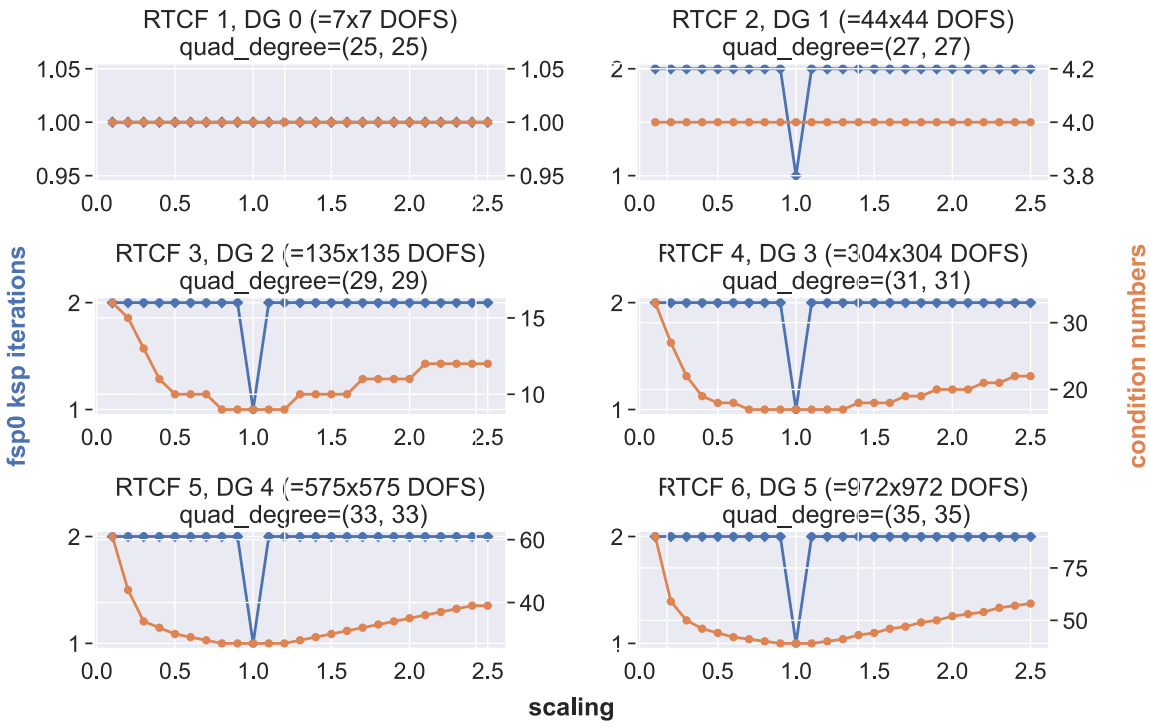


Figure 4.6: Scaled cells: Velocity mass solver iterations

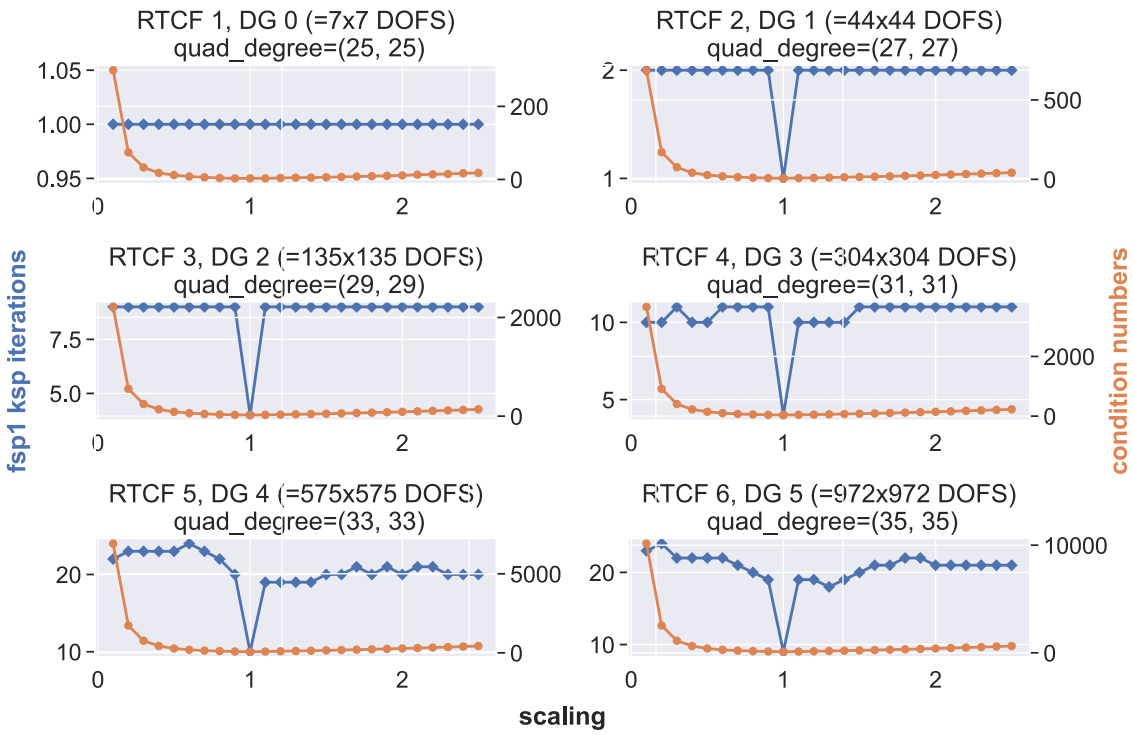


Figure 4.7: Scaled cells: Schur complement solver iterations

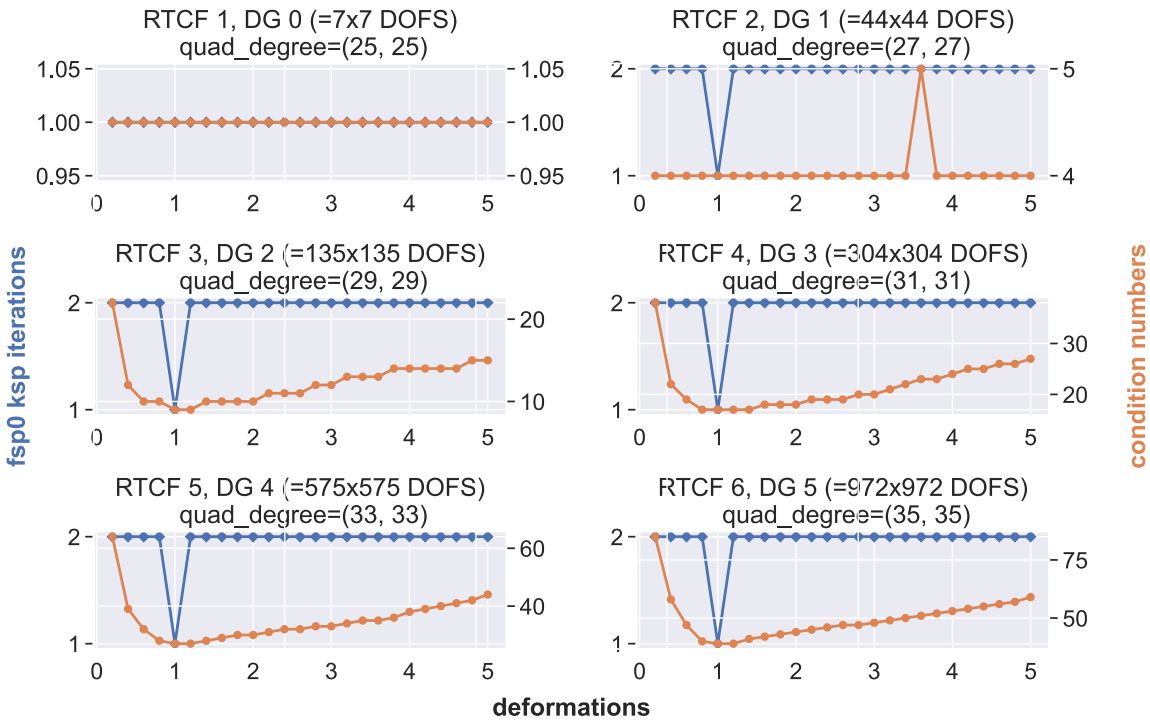


Figure 4.8: Affinely deformed cells: Velocity mass solver iterations

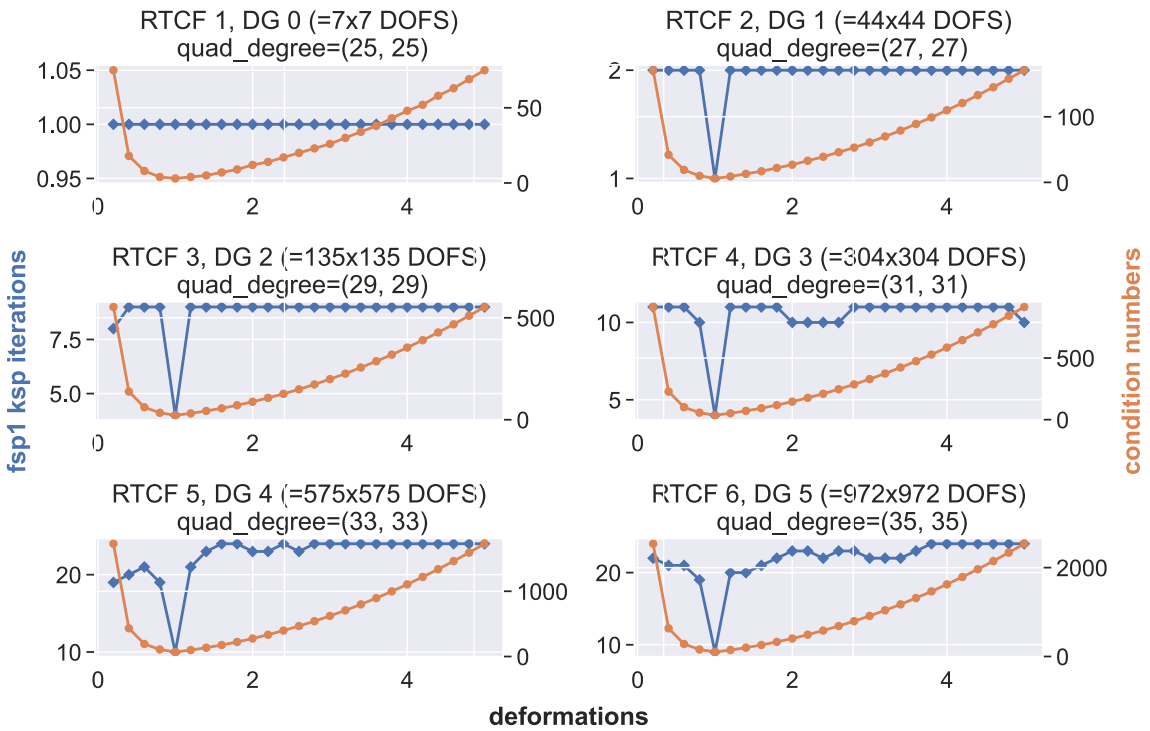


Figure 4.9: Affinely deformed cells: Schur complement solver iterations

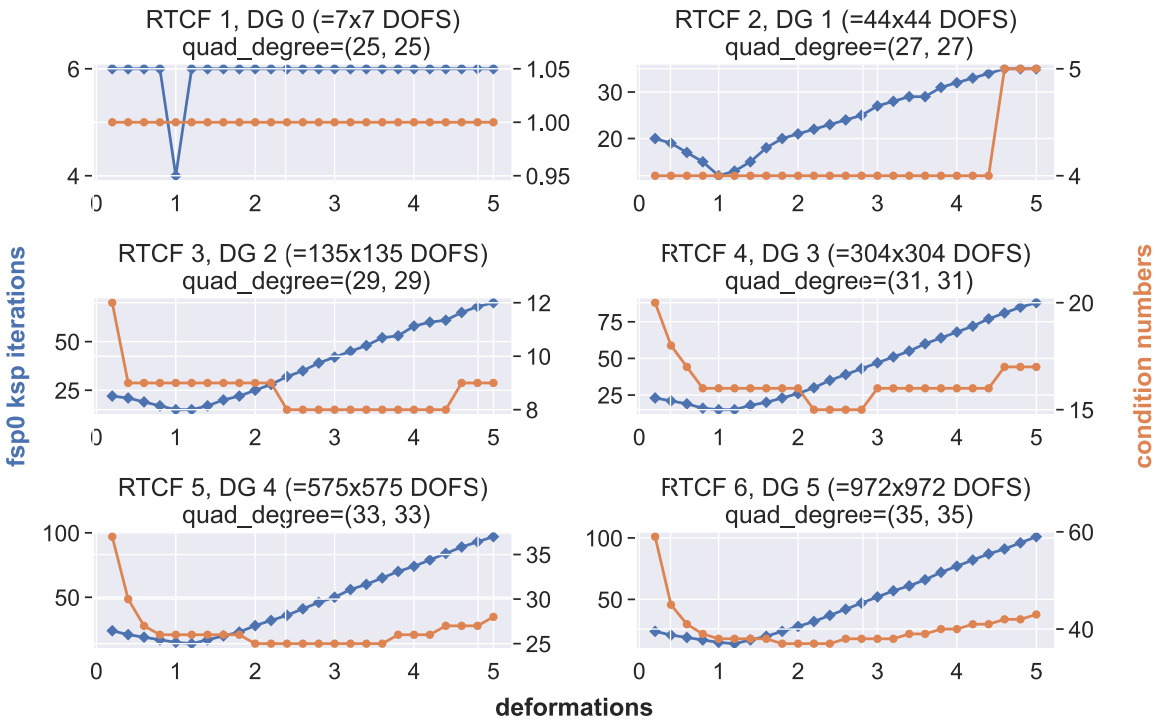


Figure 4.10: Non-affinely deformed cells: Velocity mass solver iterations

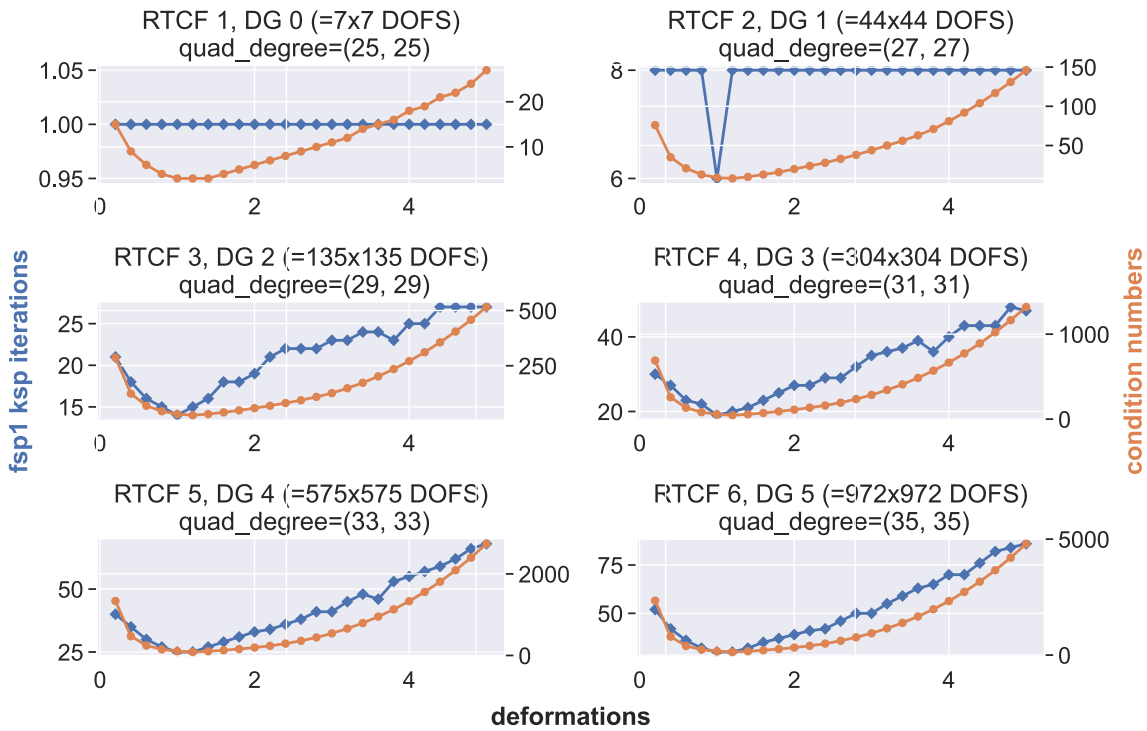


Figure 4.11: Non-affinely deformed cells: Schur complement solver iterations

4.1.2.3 A local solver for the mixed matrix in the trace solve

Since the local, mixed matrix $\begin{pmatrix} \mathbf{A}_{00}\,\mathbf{A}_{01} \\ \mathbf{A}_{10}\,\mathbf{A}_{11} \end{pmatrix}$ is indefinite, just like the original, global operator $\mathscr{A}$ of the hybridised problem, a Schur reduction can be applied in a similar way. The explicit form of the inverse of the Schur decomposition of the cell-local inverse of the matrix on the first line of the page is denoted by **D** (for decomposition not for diagonal).

$$\mathbf{D}^K := \begin{bmatrix} \mathbf{I} & -\mathbf{A}_{00}^{K^{-1}}\mathbf{A}_{01}^K \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{A}_{00}^{K^{-1}} & \mathbf{0} \\ \mathbf{0} & \mathbf{S}_p^{K^{-1}} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{A}_{10}^K\,\mathbf{A}_{00}^{K^{-1}} & \mathbf{I} \end{bmatrix} \tag{4.9}$$

$\mathbf{S}_p^K$ (= $\mathbf{S}_{11}$ in equation (4.7)) is the inner Schur complement, which is formed with respect to the scalar variable p.

$$\mathbf{S}_p^K := \mathbf{A}_{11}^K - \mathbf{A}_{10}^K\,\mathbf{A}_{00}^{K^{-1}}\mathbf{A}_{01}^K \tag{4.10}$$

Replacing the inverse of the mixed matrix in the equations (4.3) and (4.4) for the trace system solve with the inverse of the Schur complement decomposition **D** yields the following set of equation.

$$\mathbf{S}_\lambda \Lambda = \mathbf{E} \tag{4.11}$$

$$\text{with} \quad \mathbf{S}_\lambda^K = \mathbf{A}_{22}^K - \left[\mathbf{A}_{20}^K, \mathbf{A}_{21}^K\right] \begin{bmatrix} \mathbf{A}_{00}^K & \mathbf{A}_{01}^K \\ \mathbf{A}_{10}^K & \mathbf{A}_{11}^K \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{A}_{02}^K \\ \mathbf{A}_{12}^K \end{bmatrix} = \mathbf{A}_{22}^K - \left[\mathbf{A}_{20}^K, \mathbf{A}_{21}^K\right] \mathbf{D}^K \begin{bmatrix} \mathbf{A}_{02}^K \\ \mathbf{A}_{12}^K \end{bmatrix} \tag{4.12}$$

$$\text{and} \quad \mathbf{E}^K = \mathbf{F}_2^K - \left[\mathbf{A}_{20}^K, \mathbf{A}_{21}^K\right] \begin{bmatrix} \mathbf{A}_{00}^K & \mathbf{A}_{01}^K \\ \mathbf{A}_{10}^K & \mathbf{A}_{11}^K \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{F}_0^K \\ \mathbf{F}_1^K \end{bmatrix} = \mathbf{F}_2^K - \left[\mathbf{A}_{20}^K, \mathbf{A}_{21}^K\right] \mathbf{D}^K \begin{bmatrix} \mathbf{F}_0^K \\ \mathbf{F}_1^K \end{bmatrix} \tag{4.13}$$

After introduction of the inner Schur complement decomposition no mixed matrices are inverted anymore, only the non-mixed matrices $\mathbf{A}_{00}^K$ and $\mathbf{S}_p^K$. Note that the same matrices are involved in the reconstruction calls in equation (4.5) and (4.6). Therefore, a byproduct of the nesting of the Schur complements is that the local solvers from the reconstructions, can also be reused in the trace solve.

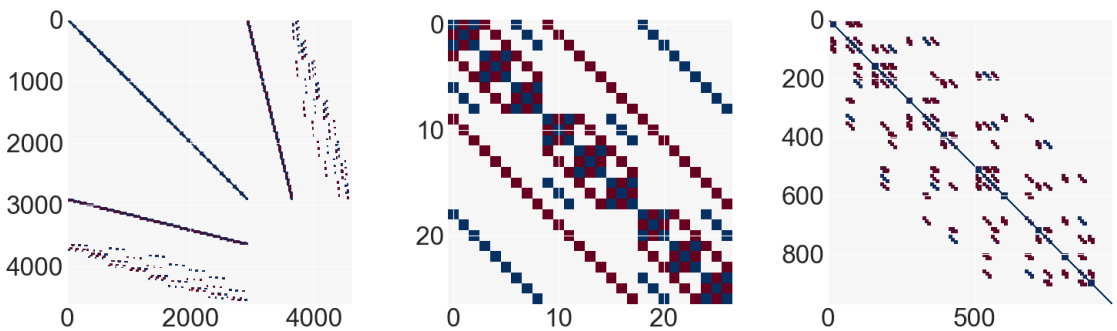


Figure 4.12: Structure of the hybridised mixed Poisson matrix $\mathscr{A}$ from equation (4.1), outer Schur complement $\mathbf{S}_\lambda$ from equation (4.2), (4.11), and inner Schur complement $\mathbf{S}_{11}$, $\mathbf{S}_p$ from equation (4.7) and (4.10)

4.2 Code infrastructure for local matrix-free solvers

Before the work presented in this thesis, Firedrake already had some support for matrix-free methods. It also had support for local linear algebra operations and for vectorisation (on development branches). All of these features were supported standalone. The missing pieces of code infrastructure for vectorisation and matrix-free methods for FEM involving local linear algebra operations were, first of all, a compiler which produces code for Slate expressions in the same language as the vectorised assembly of FE tensors as in the work in [28]. The details about the new Slate compiler can be found in chapter 3.2. Secondly, fully matrix-free methods were limited to the assembly of FE one-forms (matrix applications on vectors) and while they were supported for local linear algebra operations on tensors of those forms on a *global* level, they were not on a *local* level. For example inverses and solves on the local assembly tensors were performed in a matrix-explicit manner. The new code infrastructure to support fully matrix-free FEM which require local linear algebra operations is presented in this section.

The contributions of new infrastructure can be separated into multiple parts:

1. The optimisation passes for resorting the local linear algebra expressions into the most efficient order, see section 4.2.1

2. The extensions to the Slate and GEM language and further translations to Loo.py to represent and generate code for

   (a) local actions as a replacement for explicit matrix-vector products, see section 4.2.2

   (b) local matrix-free solvers, see section 4.2.3

   (c) local matrix-free preconditioning, see section 4.2.4

4.2.1 A series of Slate optimisation passes

Slate users can define local linear algebra expressions in their preferred form without considering the order of the operations. It is left up to an optimisation pass[1] to rewrite the expressions into their most efficient form.

[1] In our case, an optimisation or compiler pass is an iteration through the nodes in the abstract syntax tree (AST) transforming each node according to some rules. The AST is the data structure which represents the local linear algebra expression.

For high-order FEM, the most efficient order of the operations is the one where no matrices are assembled, if possible, or at least only the blocks of the matrices used for further operations are assembled. The term rewriting pass exploits the equivalence of various linear algebra expressions, building on distributive, commutative, associative, transpose laws and more.

An easy optimisation example would be $(A+B)\cdot f \rightarrow A\cdot f + B\cdot f$ with A and B being matrices and f being a vector. This transformation avoids the matrix addition of A and B before a matrix-vector product in favour of doing two matrix-vector products and then adding them together. While the term rewriting pass eliminates (or reduces) the cost of storing more than one-dimensional temporaries, it increases the number of operations required to assemble a Slate expression locally.

The new optimiser not only takes care of rewriting the order of the expression for multiplications and blocks, but also removes double transposes and could potentially replace tensors multiplied with their inverse with identity matrices (the latter is not implemented yet).

4.2.1.1 Slate optimisations as term rewriting systems

The Slate optimiser can be understood as a term rewriting system [79]. A term rewriting system consists of a set of terms defined over an alphabet and reduction rules on those terms. Each rule defines a set of rewrites with help of substitution maps.

The Slate term rewriting system can be defined by the alphabet $\Sigma = \{\mathbf{T}, \mathbf{F}\}$ of the Slate language, which contains the tensors and their operations, and a set of rules $\mathbf{R} = \{R_0, \ldots, R_n\}$ on the terms of the alphabet. The Slate language is defined in section 3.1.1.

4.2.1.2 Multiplication optimisation pass (MOP)

The multiplication optimisation pass serves the purpose of reducing the amount of intermediate matrix-shaped temporaries required to build up Slate expressions. A multiplication with a vector is pushed as far inwards as possible. The set of rules in the TRS for pushing multiplications inside an expression is defined in table 4.2, in application to the Slate alphabet. Some examples of the application of reduction rules are presented in example 4.2.1 and example 4.2.2.

Axiom	Rule			Applicability condition
Distributivity	R_1:	$(T_1+T_2)\cdot C$	$\rightarrow$ $T_1\cdot C+T_2\cdot C$	
Associativity	R_2:	$(T_1\cdot T_2)\cdot C$	$\rightarrow$ $T_1\cdot(T_2\cdot C)$	
	R_3:	$-T_1\cdot C$	$\rightarrow$ $-(T_1\cdot C)$	
Transpose laws	R_4:	$(T_1+T_2)^T$	$\rightarrow$ $T_1^T+T_2^T$	
	R_5:	$(T_1\cdot T_2)^T$	$\rightarrow$ $T_2^T\cdot T_1^T$	
	R_6:	$(T_1^T)^T$	$\rightarrow$ T_1	
Inverse laws	R_7:	$(T_1\cdot T_2)^{-1}$	$\rightarrow$ $T_2^{-1}\cdot T_1^{-1}$	if T_1 and T_2 are invertible
	R_8	$(T_1)^{-1}\cdot C$	$\rightarrow$ $T_1\backslash C$	if T_1 is invertible

Table 4.2: Reduction rules in the TRS of the Slate multiplication optimisation pass

Ex. 4.2.1. *Application of associativity on* $T_1\cdot T_2\cdot T_3\cdot C$

Let $T_1, T_2, T_3, C \in \mathbf{T}$, where $T_i \in \mathbb{R}^{10\times 10}$ are tensors and $C \in \mathbb{R}^{10}$ a known function in V_h and $\cdot$ be a dot product. Let the original expression be $T_1\cdot T_2\cdot T_3\cdot C$.

$$T_1\cdot T_2\cdot T_3\cdot C = (T_1\cdot T_2\cdot T_3)\cdot C \rightarrow_{R_2} (T_1\cdot T_2)\cdot(T_3\cdot C) \tag{4.14}$$

$$\rightarrow_{R_2} T_1\cdot(T_2\cdot(T_3\cdot C)) \tag{4.15}$$

The corresponding transformation of the DAG can be seen in figure 4.13.

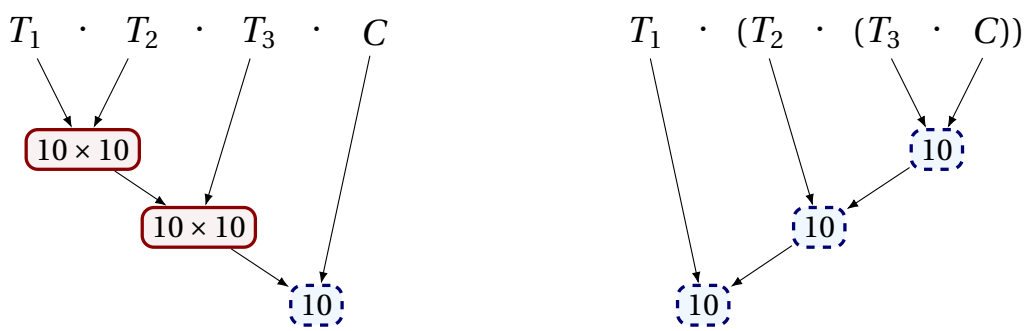


Figure 4.13: The temporaries needed for operations involved in a simple linear algebra expression before (left) and after (right) the multiplication optimisation. Red, continuously lined boxes indicate matrix shaped temporaries and blue, striped boxes indicate vector shaped temporaries.

Ex. 4.2.2. *Optimisation of* $(T_1 - T_2 \cdot T_1^{-1} \cdot T_3) \cdot C$

Let $T_1, T_2, T_3, C \in \mathbf{T}$, where $T_i \in \mathbb{R}^{10\times 10}$ are tensors and $C \in \mathbb{R}^{10}$ a known function in V_h and $\cdot$ be a dot product. Let the original expression be $(T_1 - T_2 \cdot T_1^{-1} \cdot T_3) \cdot C$, which is close to a Schur complement.

$$(T_1 - T_2 \cdot T_1^{-1} \cdot T_3) \cdot C \rightarrow_{R_1} T_1 \cdot C - (T_2 \cdot T_1^{-1} \cdot T_3) \cdot C \tag{4.16}$$

$$\rightarrow_{R_2^2} T_1 \cdot C - T_2 \cdot T_1^{-1} \cdot (T_3 \cdot C) \tag{4.17}$$

$$\rightarrow_{R_8} T_1 \cdot C - T_2 \cdot T_1 \backslash (T_3 \cdot C) \tag{4.18}$$

The corresponding transformation of the DAG can be seen in figure 4.14.

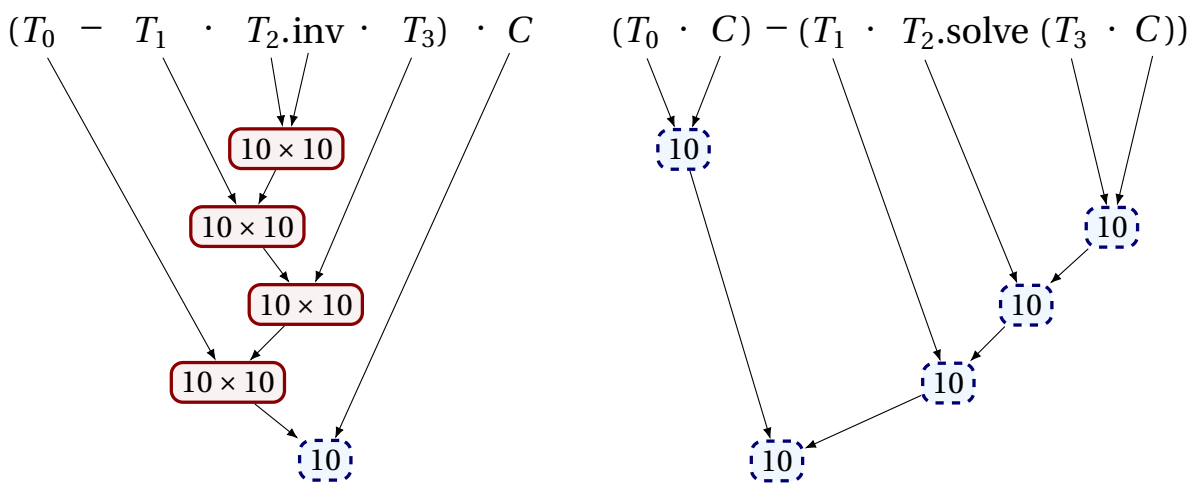


Figure 4.14: The temporaries needed for operations involved in a Schur complement like expression before (left) and after (right) the multiplication optimisation. Red, continuously lined boxes indicate matrix shaped temporaries and blue, striped boxes indicate vector shaped temporaries.

After the optimisation pass there are no matrix-shaped temporaries needed for the operations on the terminals anymore, as can be seen for example in the visualisation of the DAGs in 4.14. Though, without some additional work, the temporaries for the terminals in the expression still need to initialised with some data in a first step, and the terminals can still be matrix-shaped .

It is possible to avoid building the matrix-shaped terminals by replacing multiplications with actions. Explanations on that can be found in one of following sections 4.2.2.

4.2.1.3 Block optimisation pass (BOP)

Blocks in Slate are used to index into tensors of forms defined on mixed function spaces. For example if there is a `form` defined on $W \times W$ with $W = U \times V$ and one would build the tensor `A = Tensor(form)`, then one can get the contributions of the $U \times U$ block by asking for `A._blocks[0,0]`.

Previously, the assembly of the kernels corresponding to the blocks happened in the order: assemble first, index later. Firedrake assembled the local TSFC kernels (corresponding to local assembly A in the example above) and then indexed into the result of the local assembly kernel in a Slate kernel. Afterwards, further linear algebra operations can be applied. The block optimisation pass changes the order of assembly and indexing to: first indexing the FE form, then assembling the indexed form locally with TSFC. This is important, because it means that not the whole local tensor is assembled when only parts of it are needed.

	Rule			
Distributivity	R_1 :	$\text{Block}((T_1 + T_2), \text{idx})$	$\rightarrow$	$\text{Block}(T_1, \text{idx}) + \text{Block}(T_2, \text{idx})$
Associativity	R_2 :	$\text{Block}(-T_1, \text{idx})$	$\rightarrow$	$-\text{Block}(T_1, \text{idx})$
Transpose laws	R_3 :	$\text{Block}(T_1, \text{idx})^T$	$\rightarrow$	$\text{Block}(T_1, \text{reverse}(\text{idx}))$
Terminal tensors	R_4 :	$\text{Block}(T_1, \text{idx})$	$\rightarrow$	$T_{\text{idx},1}$
	R_5 :	$\text{Block}(V_1, \text{idx})$	$\rightarrow$	$V_{\text{idx},1}$
Nested Blocks	R_6 :	$\text{Block}(\text{Block}(T_1, \text{idx}_1), \text{idx}_2)$	$\rightarrow$	$\text{Block}(T_1, \text{idx}_2[\text{idx}_1])$
Other Nodes	R_7 :	$\text{Node}(T_1, \text{idx})$	$\rightarrow$	$\text{Node}(\text{T}_1, \text{idx})$

Table 4.3: Reduction rules in the TRS of the Slate block optimisation pass

The change of order in the operations is achieved by pushing the blocking node as far as possible inside the linear algebra expression. For the corresponding set of rules see table 4.3. Due to distributivity and associativity, blocks can simply be pushed into additions and negations. In order to push into a Transpose node the indices need to be reversed. Blocks are not pushed into factorisations, inverses, solves and multiplications. When a Block is wrapped around a terminal tensor or vector, the rule R_4 and R_5 represent that a new tensor or vector is build from the indexed variational form or coefficient. Nested blocking is covered by the rule R_6.

The block optimisation, defined in table 4.3, can be neatly turned into code with the `singledispatch` functionality of Python and the `Memoizer` functionality provided by TSFC, see the listing in figure 4.15. The code neatly reflects the rules in the block optimisation pass, each rule has one function.

```
def push_block(expression):
    mapper = MemoizerArg(_push_block)
    ret = mapper(expression, ())
    return ret


@_push_block.register(sl.Transpose)
def _push_block_transpose(expr, self, indices):
    """Indices of the Blocks are transposed if Block is pushed into a
   Transpose."""
    return sl.Transpose(*map(self, expr.children, repeat(indices[::-1])))


@_push_block.register(sl.Add)
@_push_block.register(sl.Negative)
def _push_block_distributive(expr, self, indices):
    """Distributes Blocks for these nodes"""
    return type(expr)(*map(self, expr.children, repeat(indices)))


@_push_block.register(sl.Factorization)
@_push_block.register(sl.Inverse)
@_push_block.register(sl.Solve)
@_push_block.register(sl.Mul)
def _push_block_stop(expr, self, indices):
    """Blocks cannot be pushed further into this set of nodes."""
    expr = type(expr)(*map(self, expr.children, repeat(tuple())))
    return Block(expr, indices) if indices else expr


@_push_block.register(sl.AssembledVector)
def _push_block_assembled_vector(expr, self, indices):
    """Turns a Block on an AssembledVector into the  specialized node
   BlockAssembledVector."""
   return (BlockAssembledVector(expr._function, Block(expr, indices),
   indices) if indices else expr)


@_push_block.register(sl.Block)
def _push_block_block(expr, self, indices):
    """Inlines Blocks into each other.
    The second time round, (0,0) is stored in indices and the slices are
   in expr._indices.
    So in the following line we basically say indices = ((0,1,2)[0],
   (0,1,2)[0])"""
    reindexed = tuple(big[slice(small[0], small[-1]+1)]
                      for big, small in zip(expr._indices, indices))
    indices = expr._indices if not indices else reindexed
    block, = map(self, expr.children, repeat(indices))
    return block

... (further functions) ...
```

Figure 4.15: Code of the block optimisation. Note that the optimisation of nodes introduced in later chapters is not shown here. `BlockAssembledVector` is a new node to represent blocks on vectors.

4.2.2 Replacing multiplication by Actions

An action is an assembly of a 1-form, see definition 2.3.1. The benefits of using actions instead of the multiplications are the same as for the term-rewriting Slate optimisation passes. The reduction/removal of the load bottleneck in high order FEM. Actions are used to avoid building matrix-shaped temporaries because of their rank-reducing nature, refer to definition 2.3.1.

There are two levels where actions can be applied in the FEM stack: one is the local assembly, and one is the global assembly, see chapter 2.1.3. The only difference between them is the size of the operators the actions work on. The matrices involved in the global assembly are bigger than the matrices involved in the local assembly. While the size of the matrices for the former is depending on the amount of elements in the mesh, the latter depends on the polynomial order of the chosen discretisation. While there are usually enough elements in the mesh to make global actions worthwhile, the order of the approximation must be high enough for local actions to have a positive effect on the performance of the FEM. As mentioned previously, matrix-free methods generally store and load less data than matrix-explicit approaches, but need more FLOPS. That is a trade-off, which only pays off for higher-order FEMs.

In the setting of the Firedrake framework, another difference between the global and local actions is the component responsible for expressing the action. While global actions are expressed at the UFL level and work on variational forms, local actions are introduced at Slate level, meaning they are operations on Slate tensors. The infrastructural changes I introduced are visualised in figure 4.18. The introduction of an Action node in Slate is followed by a translation into a GEM Action node and further into a Loo.py `CallInstruction`. These `CallInstructions` are then resolved in a separate compiler pass and replaced by the TSFC assembly kernels of vectors (matrix-free application of forms).

One difficulty for local actions is that one action depends on the results of another. Previously, the Slate compiler filled all terminal tensors and vectors of the Slate expression kernel upfront (with `subkernel0` and `subkernel1` in line 19 and 20 in figure 4.16), before executing the linear algebra operations on them. That was only possible, because tensors representing UFL forms were not dependent on former Slate operations. With the introduction of Actions, this has changed. The order of the dependencies needs to be respected. One assembly kernel may depend on the results of another or the result of linear algebra operations on another, so that tensors cannot be filled through local assembly kernels upfront anymore. This stacking behaviour becomes more clear in the generated code in figure 4.17 using the new infrastructure for local actions. The function `subkernel1` takes the result of the local assembly kernel `subkernel0` as a parameter.

```
static void wrap_slate_loopy_knl_0(double *__restrict__ output,
                                   double const *__restrict__ coords,
                                   double const *__restrict__ w_0){

  /* Matrix shaped temporaries */
  double T1[10 * 10];
  double T2[10 * 10];
  double C[10];
  for (int32_t id = 0; id <= 9; ++id)
    for (int32_t id_0 = 0; id_0 <= 9; ++id_0)
      T1[10 * id + id_0] = 0.0;
  for (int32_t id_1 = 0; id_1 <= 9; ++id_1)
    for (int32_t id_2 = 0; id_2 <= 9; ++id_2)
      T2[10 * id_1 + id_2] = 0.0;
  for (int32_t id_3 = 0; id_3 <= 9; ++id_3)
    C[id_3] = w_0[id_3];

  /* Matrix-explicit local assembly calls */
  subkernel0_cell_to__cell_integral_otherwise(&(T1[0]), &(coords[0]));
  subkernel1_cell_to__cell_integral_otherwise(&(T2[0]), &(coords[0]));

  /* Call to linear algebra operations */
  slate_loopy_knl_0(&(output[0]), &(T1[0]), &(T2[0]), &(C[0]));
}



static void slate_loopy_knl_0(double *__restrict__ output,
                              double const *__restrict__ T1,
                              double const *__restrict__ T2,
                              double const *__restrict__ C)

  /* Temporary declaration */
  double t0[10];

  /* Linear algebra operations */
  for (int32_t i = 0; i <= 9; ++i)
  {
    for (int32_t i_0 = 0; i_0 <= 9; ++i_0)
      t0[i_0] = 0.0;
    for (int32_t i_1 = 0; i_1 <= 9; ++i_1)
      for (int32_t i_2 = 0; i_2 <= 9; ++i_2)
        t0[i_2] = t0[i_2] + T1[10 * i + i_1] * T2[10 * i_1 + i_2];
    for (int32_t i_3 = 0; i_3 <= 9; ++i_3)
      output[i] = output[i] + t0[i_3] * C[i_3];
  }
}
```

Figure 4.16: Non-optimised Slate code for $(T_1 \cdot T_2) \cdot C$. More detailed explanations of the code are given in the text.

```
static void slate_loopy_knl_3(double *__restrict__ output,
                              double const *__restrict__ coords,
                              double const *__restrict__ w_0){

  /* Vector shaped temporaries */
  double A0[10];
  double A1[10];
  double C[10];
  for (int32_t id_7 = 0; id_7 <= 9; ++id_7)
    C[id_7] = w_0[id_7];

  /* 1st action  */
  for (int32_t id = 0; id <= 9; ++id)
    A0[id] = 0.0;
  subkernel0_cell_to__cell_integral_otherwise(&(A0[0]), &(coords[0]), &(C
    [0]));
  for (int32_t id_3 = 0; id_3 <= 9; ++id_3)
    A1[id_3] = 0.0;]

  /* 2nd action */
  subkernel1_cell_to__cell_integral_otherwise(&(A1[0]), &(coords[0]), &(A0
    [0]));
  for (int32_t i_7 = 0; i_7 <= 9; ++i_7)
    output[i_7] = output[i_7] + A1[i_7];
}
```

Figure 4.17: Optimised Slate code for $(T_1 \cdot T_2) \cdot C$. More detailed explanations of the code are given in the text.

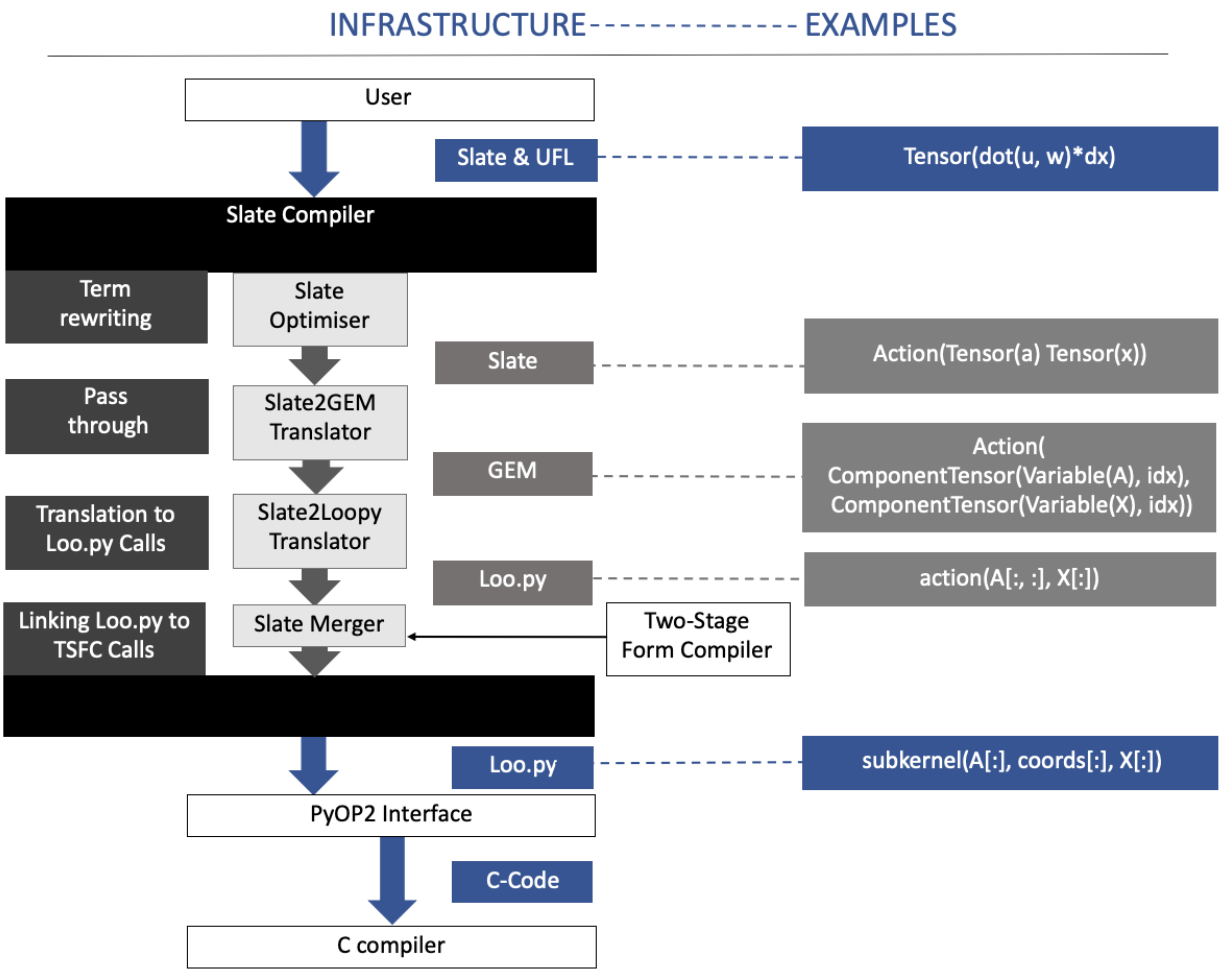


Figure 4.18: New infrastructure to support local actions in Firedrake. Black and gray colors indicate modified infrastructure as part of this project. The Slate compiler in black consists of three columns. The middle column shows different pieces of the Slate compiler pipeline. Left of that is a short explanation about how the corresponding compiler piece treats an action and right of the language produced by the compiler piece is indicated.

4.2.3 A local matrix-free solver and TensorShell nodes

Replacing all multiplications in the local linear algebra expressions by actions is not the end of the story to achieve high performance for high order FEM. There are still the non-trivial algebra operations of an inverse and a solve, which need to be rewritten so that no matrix-shaped temporaries are built anymore. Inverses can be turned into solves through $A.\text{inv} \cdot b = A.\text{solve}(b)$. The solve nodes remain to be changed. Matrix-explicit solve (and inverse) nodes are placeholders which are passed through the compiler stack until in the final step they are translated to kernels, which consist of a set of instructions in C, see figure 3.3, for an LAPACK factorisation and solve. Replacing these C kernels with matrix-free, iterative loopy solve kernels completes the process of rewriting a matrix-explicit expression into its matrix-free form.

The matrix-free solver presented here is an unpreconditioned Conjugate Gradient method. If preconditioners are supplied, the algorithm switches automatically to the preconditioned algorithm. There are some technical details to be considered. As previously mention, the matrix-free solver is implemented in Loo.py. Calling to PETSc solver would cause to much overhead since these solves are called on every single cell in the mesh. Since Loo.py does not support while loops yet, the iteration runs to a maximum iteration supplied through the user or N times, but has a stop criterion to abort when convergence is achieved. The iteration is also aborted when the projector is close to zero. Pseudocode for the algorithm can be found in algorithm 3.

Algorithm 3: A matrix-free solver is CG method for $\mathbf{A} \in \mathbb{R}^{N\times N}$, $\mathbf{b} \in \mathbb{R}^{N}$.

The function $\text{action}_A(\mathbf{f})$ calculates the matrix-free application of $\mathbf{A}$ on $\mathbf{f}$.

Data: action_A, $\mathbf{b}$

Result: x

$\mathbf{x_0} \rightarrow$ Initialise with $\mathbf{b}$;

$\mathbf{A\_on\_x_0} \rightarrow$ Call local kernel to calculate $\mathbf{A\_on\_x_0} = \text{action}_A(\mathbf{x_0})$;

$\mathbf{r_0} \rightarrow$ Calculate initial residual with $\mathbf{r_0} = \mathbf{A\_on\_x_0} - \mathbf{b}$;

$\mathbf{p_0} \rightarrow$ Calculate initial projector with $\mathbf{p_0} = -\mathbf{r_0}$;

$n_k \rightarrow$ Calculate norm of the initial residual with $n_k = \sum_i \mathbf{r_0}[i]^2$

for $k \in \{0, \ldots, N\}$ **do**

  $\mathbf{A\_on\_p_k} \rightarrow$ Call local kernel to calculate $\text{action}_A(\mathbf{p_k})$;

  $\text{p\_on\_A\_on\_p}_k \rightarrow$ Calculate norm of the i-th residual with

  $\text{p\_on\_A\_on\_p}_k = \sum_i \mathbf{p_k}[i] \cdot \mathbf{A\_on\_p_k}[i]$;

  if p_on_A_on_p_k is close to zero abort;

  $\alpha \rightarrow$ Calculate the scaling of the search direction with

  $\alpha = n_k / \text{p\_on\_A\_on\_p}_k$;

  $\mathbf{x_{k+1}} \rightarrow$ Calculate solution by moving coordinates in the direction of the

  scaled projector by $\mathbf{x_{k+1}} = \mathbf{x_k} + \alpha \cdot \mathbf{p_k}$;

  $\mathbf{r_{k+1}} \rightarrow$ Calculate new residual with $\mathbf{r_{k+1}} = \mathbf{r_k} + \alpha \cdot \mathbf{A\_on\_p_k}$;

  $n_{k+1} \rightarrow$ Calculate norm of the new residual with $n_{k+1} = \sum_i \mathbf{r_{k+1}}[i]^2$;

  if converged abort;

  $\beta \rightarrow$ Calculate ratio of new and old residual with $\beta = n_{k+1}/n_k$;

  $n_k \rightarrow$ Reset norm $n_k = n_{k+1}$;

  $\mathbf{p_{k+1}} \rightarrow$ Calculate new projector with $\mathbf{p_{k+1}} = \beta \mathbf{p_k} - \mathbf{r_k}$;

return the solution x;

In the case that the code is generated to support vectorisation, the conditional to abort the loop is explicitly vectorised through Loo.py. All solves on the cells in the batch abort if one of them is converged. In order to still get the wished-for accuracy a lower tolerance should be chosen.

Algorithm 4: A Slate kernel for a matrix-explicit solve on a non terminal tensor:
The kernel calculates $(\mathbf{A_1} + \mathbf{A_2})$.solve($\mathbf{b}$).

Data: form_1, form_2, coeff_3
Result: $\mathbf{x}$

```
A1, A2 → local assembly of form1,2;
b → fill with data from coeff3;
A = A1 + A2;
Function Solve kernel():
    solve Ax = b;
more Slate operations;
```

Algorithm 5: A Slate kernel for a matrix-free solve on a non terminal tensor
The kernel calculates $(\mathbf{A_1} + \mathbf{A_2})$.matfree_solve($\mathbf{b}$).

Data: form_1, form_2, coeff_3
Result: $\mathbf{x}$

```
b → fill with data from coeff3;
A1,A2 → initialise;
Function Matrix-free solve kernel():
    Function Tensorshell kernel():
        return action(A1, b) + action(A2, b);
    Calculate initial residual r0 and initial projector p0;
    while not converged do
        Function Tensorshell kernel():
            return action(A1, p0) + action(A2,p0);
        more solve operations;
more Slate operations;
```

The tensor $\mathbf{A}$ that is solved for in $\mathbf{Ax} = \mathbf{b}$ is not necessarily a terminal tensor. There may be an addition involved, for example consider $(\mathbf{A_1} + \mathbf{A_2})$.solve($\mathbf{b}$). Depending on whether the matrix $\mathbf{A}$ is terminal or not, the action calls in line 2 and 7 of the CG method in algorithm 3 are replaced by different function calls.

- When the matrix in the encountered action is terminal, it can be directly replaced by a local assembly call.
- Otherwise, the linked instruction is a call to a linear algebra kernel. A visualisation of the nesting of kernels for this case is given in algorithm 5

Note that for complicated expressions this can yield many-nested function calls.

In order to support this behaviour another node was introduced into Slate, called a `TensorShell`, which behaves like a terminal tensor even though it is only a placeholder for operations on tensors. A `TensorShell` is the equivalent of a PyOP2 `ImplicitMatrix`, just in a local assembly kernel, or PETSc `MatShell` in a later stage. The operations represented by a `TensorShell` are not translated within the first compiler pass, but later in the merging step of the Slate compiler, where actions are resolved. Effectively, the new `TensorShell` moves the evaluation of a set of operations to where enough information is available. For example $(\mathbf{A_1} + \mathbf{A_2})$ from the previous example is moved from an outer kernel (a Slate wrapper kernel) to an inner kernel (the matrix-free solve), as be seen in the comparison of algorithms 4 and 5.

4.2.4 Matrix-free preconditioning of the local solvers

4.2.4.1 User-defined preconditioners for local solvers

As mentioned in the previous sections, the local solves can be sped up by using local preconditioners. Just to give a motivation, consider that in a discontinuous space the Schur complement $\mathbf{S}_\mathbf{p}^K$ of the mixed Poisson problem is spectrally equivalent to a Laplacian, so that it is a good idea to use the DG-Laplacian, see figure 4.19 for the implementation, as an approximation to the Schur complement and use it as a preconditioner for the matrices of the scalar variable, the pressure p. I have introduced new infrastructure in Firedrake, so that users can define a so called `AuxiliaryOperator` to declare a form, written in UFL, which is not a part of the original problem, for the local preconditioner. This is not limited to the example of DG-Laplacian. The operator is then passed through the stack and used as a preconditioner to the local solve.

Note that only left preconditioning is considered such that any of the local solvers, which solve for the unknown $\mathbf{x}$ with system matrix $\mathbf{A}$ and right-hand side $\mathbf{b}$, are changed from an unpreconditioned solve $\mathbf{Ax} = \mathbf{b}$ into a with $\mathbf{P}$ preconditioned system $\mathbf{P}^{-1}\mathbf{Ax} = \mathbf{P}^{-1}\mathbf{b}$. When a preconditioner is specified, the preconditioned algorithm for the local CG solve is used.

Let the equation of the pressure reconstruction (2.37) from example 2.2.2 be considered, where the Schur complement can be preconditioned with an `AuxiliaryOperator`. Denote the right hand side of (2.37) as $\mathbf{R}_\mathbf{p}^\mathbf{K} := \mathbf{F}_1^\mathbf{K} - \mathbf{A}_{10}^\mathbf{K}\left(\mathbf{A}_{00}^\mathbf{K}\right)^{-1}\mathbf{F}_0^\mathbf{K} - \left(\mathbf{A}_{12}^\mathbf{K} - \mathbf{A}_{10}^\mathbf{K}\left(\mathbf{A}_{00}^\mathbf{K}\right)^{-1}\mathbf{A}_{02}^\mathbf{K}\right)\Lambda^\mathbf{K}$ and the user-defined local preconditioner as $\mathbf{P}_\mathbf{p}^\mathbf{K}$ and $\mathbf{S}_\mathbf{p}^\mathbf{K} = \mathbf{S}_{11}$
Then, for the reconstructions calls the original equations in example 2.2.2 change from equation (4.19) to (4.20).

$$\mathbf{S}_\mathbf{p}^\mathbf{K}\mathbf{p}^\mathbf{K} = \mathbf{R}_\mathbf{p}^\mathbf{K} \tag{4.19}$$

$$\Leftrightarrow \left(\mathbf{P}_\mathbf{p}^\mathbf{K}\right)^{-1}\mathbf{S}_\mathbf{p}^\mathbf{K}\mathbf{p}^\mathbf{K} = \left(\mathbf{P}_\mathbf{p}^\mathbf{K}\right)^{-1}\mathbf{R}_\mathbf{p}^\mathbf{K} \tag{4.20}$$

The presented mechanism of user-defined local preconditioners is generic and if users solve other problems, where different local preconditioners are needed, the same mechanism can be used. All that needs to be specified on the frontend, is a form for the preconditioning operator and a solver parameter.

4.2.4.2 Local preconditioners involving diagonals

Instead of, or in addition to, a user-supplied preconditioner one may wish to precondition the local solvers with Jacobi. Similar to the user-defined operators only left preconditioning is considered. The difference between user-supplied preconditioners and Jacobi preconditioners is that for the latter, in a solver environment, only the corresponding option `pc_type:jacobi` has to be specified by the user. The local preconditioning operator will then be a matrix, which carries only the diagonal entries of the operator $\mathbf{A}$ it is specified for.

In the case of hybridisation on a mixed Poisson problem, Jacobi preconditioning can be applied either to the $\mathbf{A}_{00}$ velocity mass block, the Schur complement $\mathbf{S}_\mathbf{p}$ or even to the user-supplied preconditioning operator. The preconditioner can also be used to replace the operator it is applied to. For a general local solve $\mathbf{Ax} = \mathbf{b}$ and a Jacobi preconditioner applied in the usual way, the original system turns into $\mathbf{P}^{-1}\mathbf{Ax} = \mathbf{P}^{-1}\mathbf{b}$ where the preconditioner is the diagonal part of matrix $\mathbf{A}$, i.e. $\mathbf{P} = \mathrm{diag}(\mathbf{A})$.

In addition to the infrastructural changes required in the hybridisation preconditioner in Firedrake, so that the Jacobi preconditioner is applied in the right equations according to the specified solver options, the concept of diagonal local tensors had to be introduced in the Slate language in Firedrake. Further translation of Slate's `DiagonalTensor` into the languages on lower levels in the framework (in particular GEM) and code optimisations have been introduced as well. The code optimisation replaces any inverse on a diagonal tensor by a tensor, the Slate node of which is called `Reciprocal`, which carries the reciprocal values of the original tensor on its diagonal. The optimisation is introduced so that no local solves are required to build the inverse of the Jacobi preconditioning operator.

```
1  params = {'mat_type': 'matfree',
2            'ksp_type': 'preonly',
3            'pc_type': 'python',
4            'pc_python_type': 'firedrake.HybridizationPC',
5            'hybridization': {'ksp_type': 'fgmres',
6                              'pc_type': 'none',
7                              'ksp_rtol': 1e-8,
8                              'ksp_atol': 1e-90,
9                              'mat_type': 'matfree',
10                             'localsolve': {'ksp_type': 'preonly',
11                                            'pc_type': 'fieldsplit',
12                                            'pc_fieldsplit_type': 'schur',
13                                            'mat_type': 'matfree',
14                                            'fs0':{'ksp_type': 'default',
15                                                   'pc_type': 'jacobi',
16                                                   'ksp_rtol': 1e-14,
17                                                   'ksp_atol': 1e-90,
18                                                   'ksp_max_it': 5},
19                                            'fs1':{'ksp_type': 'default',
20                                                   'pc_type': 'python',
21                                                   'pc_python_type':'DGL',
22                                                   'aux_ksp_type':'preonly'
    ,
23                                                   'aux_pc_type': 'jacobi',
24                                                   'ksp_rtol': 1e-12,
25                                                   'ksp_atol': 1e-90,
26                                                   'ksp_max_it': 5}}}}
```

```
1  class DGL(AuxiliaryOperatorPC):
2      def form(self, pc, u, v):
3          W = u.function_space()
4          n = FacetNormal(W.mesh())
5          alpha = Constant(6)
6          gamma = Constant(8)
7          h = CellVolume(W.mesh())/FacetArea(W.mesh())
8          h_avg = (h('+') + h('-'))/2
9          a_dg = -(inner(grad(u), grad(v))*dx
10                  - inner(jump(u, n), avg(grad(v)))*dS
11                  - inner(avg(grad(u)), jump(v, n), )*dS
12                  + alpha/h_avg * inner(jump(u, n), jump(v, n))*dS
13                  - inner(u*n, grad(v))*ds
14                  - inner(grad(u), v*n)*ds
15                  + (gamma/h)*inner(u, v)*ds)
16         bcs = None
17         return (a_dg, bcs)
```

Figure 4.19: Solver parameters for fully matrix-free hyrbidisation. DGL is the class for the DG-Laplacian.

4.2.4.3 An example for solver options

For the motivating example of a mixed Poisson problem from section 4.1, the hybridisation process is explained in the examples 2.2.1 and 2.2.2. In Firedrake the hybridisation process is encapsulated in a preconditioner, however. The frontend user only need to define their mixed (Poisson) problem in UFL and the rest can be controlled with solver options. A full set of solver options for globally and locally matrix-free hybridisation is given as an example with a flexible GMRES solver as the global trace solver, CG methods for the local solves, a Jacobi preconditioner for the velocity mass solve and a diagonal, user-defined preconditioning operator called `DGL` for the pressure Schur complement solve. The formulation of the DG Laplacian operator is problem dependent. An example for an unextruded 2D problem is given in figure 4.19.

4.3 Results

In this chapter performance results for isolated Slate expressions (without them being used in a solver environment) are presented. There are four modes of optimisations available accessible through the infrastructure presented in this chapter. In the baseline runs, the Slate expressions are not optimised in any way. The baseline runs show the state of the art before the work presented in this thesis. Further, the new optimisation passes for reordering the Slate expressions are examined standalone and in conjunction with new local matrix-free solver and also the local preconditioner infrastructure. Further, results are shown for a vectorisation of the expression. While the performance is examined for building blocks of the hybridisation preconditioner in this section, this will be extended to an examination in a full solver environment later.

4.3.1 Software and Hardware details

For more detailed hardware details, refer to section 3.4.1. The Firedrake version for these experiments is captured in [80]. The PETSc software was built separately. It is published under [73]. The Zenodo directory for the experimental data and scripts for reproduction of the plots is [74].

	Intel(R) Xeon(R) CPU E5-2640 v3
Sockets	2
Cores per socket	8
Threads per core	2
SIMD instruction set	AVX2
Theoretical peak performance (double-precision)	665.6 GFLOP/s
Memory bandwidth performance	60 GB/s

Table 4.4: Hardware specification for the dual-socket Intel(R) Xeon(R) CPU E5-2640 v3 architecture

4.3.2 Problem setup

Performance results are shown for the building blocks of a static condensation solve for a hybridised mixed Poisson problem as presented in example 2.2.1 and 2.2.2. The problem is discretised with a hexahedral mesh. The assembly of the action (matrix-free application) of the inner (pressure) $\mathbf{S}_{\mathbf{p}}^{\mathbf{K}}$ from equation (4.10) and outer (trace variable) Schur complement $\mathbf{S}_{\lambda}^{\mathbf{K}}$ from equation (4.12) on a function $f(x_1, x_2, x_3) = x_1 \cdot (n - x_1) \cdot x_2 \cdot (n - x_2) \cdot x_3 \cdot (n - x_3)$, where the x_i are coordinates in the mesh and n is the mesh size parameter, is measured in the following experiments.

There are multiple optimisation modes of the Slate expression available. The expression order is sorted in the `expression optimised` mode, see section 4.2.1, the inverse is replaced by a local matrix-free solve in the `matfree` mode, see section 4.2.3 and the local matrix-free solve is preconditioned in the `preconditioned matfree` mode, see section 4.2.4. These modes showcase the performance of the code infrastructure introduced in this chapter. The best of those modes is further combined with the vectorisation optimisations presented in the previous chapter 3 in a `vectorised preconditioned matfree` mode.

In the `preconditioned matfree` mode the preconditioners which turned out to perform well in the investigation in section 4.1.2.2 are used. In the pressure Schur complement action only the local matrix $\mathbf{A_{00}}$ needs to be inverted. A diagonal preconditioner is used for the matrix-free solve, as explained in section 4.2.4.2. In contrast, in the trace variable Schur complement action both $\mathbf{A_{00}}$ and the pressure Schur complement $\mathbf{S}_{\mathbf{p}}^{\mathbf{K}}$ need an inversion. In the preconditioned mode, again, the solve for $\mathbf{A_{00}}$ is preconditioned with Jacobi, and further, the pressure Schur complement is preconditioned with a user-defined Laplacian operator, see section 4.2.4.1.

4.3.3 Results

DOF throughput results are presented for different optimisation modes of the inner Schur complement in figure 4.20 and for the outer Schur complement in figure 4.21. Ideally, the DOF throughput rates should be presented alongside the FLOPS throughput rates. Unfortunately, it is not feasible to count the FLOPS with the Loo.py statistics tools. The problem is that when using locally matrix-free methods, the inner loops are aborted when the local solvers converge and therefore, the Loo.py FLOP counting algorithm overestimates the amounts of FLOPS for these kernels. One possibility to overcome this is to run the local solvers to a fixed number of iterations. This will lead to either over- or underiteration in the loop for the convergence and would therefore not give a realistic performance measurement since the Schur complements need to be build exactly.

In the unoptimised, baseline case there is a significant drop in the DOF throughput for both Schur complements with increasing approximation degree. The reason for that is that the size of the local matrices in the Slate expressions increases with approximation degree. The processors are waiting longer for loading the data the bigger the matrices are. In each optimisation mode different steps are taken to work around this well-known bottleneck.

For the inner Schur complement, see figure 4.20, the throughput increases when improving the expression order so that less data needs to be stored and loaded, as expected. The DOF processing rate also drops less significantly when the Slate expression are rewritten into a matrix-free from by replacing the inverses with matrix-free solves. Further, the preconditioning improves the iteration count of the matrix-free solves and the runtime is therefore faster than for the unpreconditioned case. Finally, processing more data at a time with SIMD vectorisation speeds up the rate by around 25 percent. In the best two modes the DOF throughput stays consistent with an increase in the polynomial order for the inner Schur complement action.

For the outer Schur complement, see figure 4.21 the order of best performing modes is the same, but compared to the inner Schur complement, the DOF throughput is not as high and is also not improved as well through the optimisations. The reason for the former is that the expression is more complex, meaning more operations have to be processed so the kernels take more time to execute, while the amount of DOFs stays the same. Further, the preconditioned matrix-free optimisation does not result in a constant rate for the higher approximation degrees, because the preconditioner for the pressure variable Schur complement is not as effective as the one for the $\mathbf{A}_{00}$ block. The vectorisation does not seem to give any speedup, reasons for which are currently not understood.

For the matrix-explicit optimisation modes, the memory requirement for approximation degrees higher than what is shown in the figures 4.20 and 4.21 simply becomes prohibitive. The DOF rates for higher order approximations than that are only shown for the best two optimisation modes, see figure 4.22 and 4.23. The trends from the figures 4.20 and 4.21 are continued on higher order.

In the performance results here, the accuracy obtained by any of these methods is not shown. The expression order resorting should not result in a loss of accuracy unless there are inaccuracies due to floating point precision errors which are more pronounced in one expression order than another. Switching to the matrix-free mode, however, means that the inverses are approximated through iterative solves and an accuracy of the methods should be considered. Further, vectorised matrix-free solves use an unrolled stop criterion. The solve on all elements in a SIMD batch are aborted as soon as one of them is converging, so that the solves on some elements in the SIMD batch are underiterated. The accuracies will be examined in the final chapter.

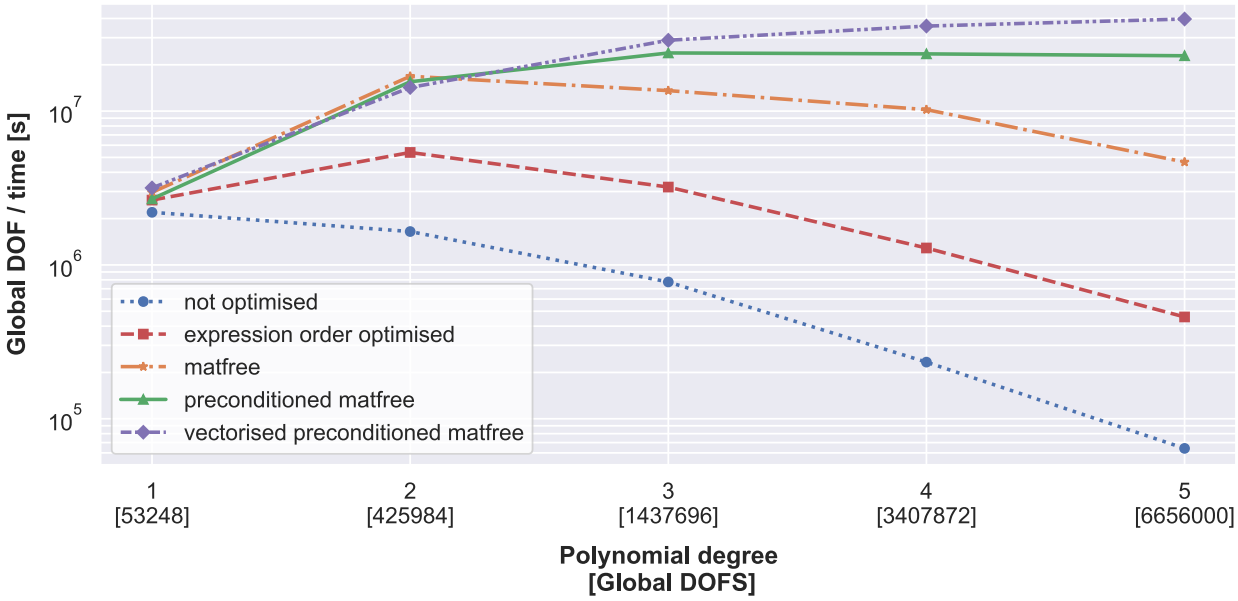


Figure 4.20: DOF throughput of the pressure Schur complement action discretised with RTCF_{p-1}-DG_p and polynomial degrees p on a $32 \times 32 \times 32$ hexahedral mesh with unit step size assembled with different optimisation modes previously presented in this chapter

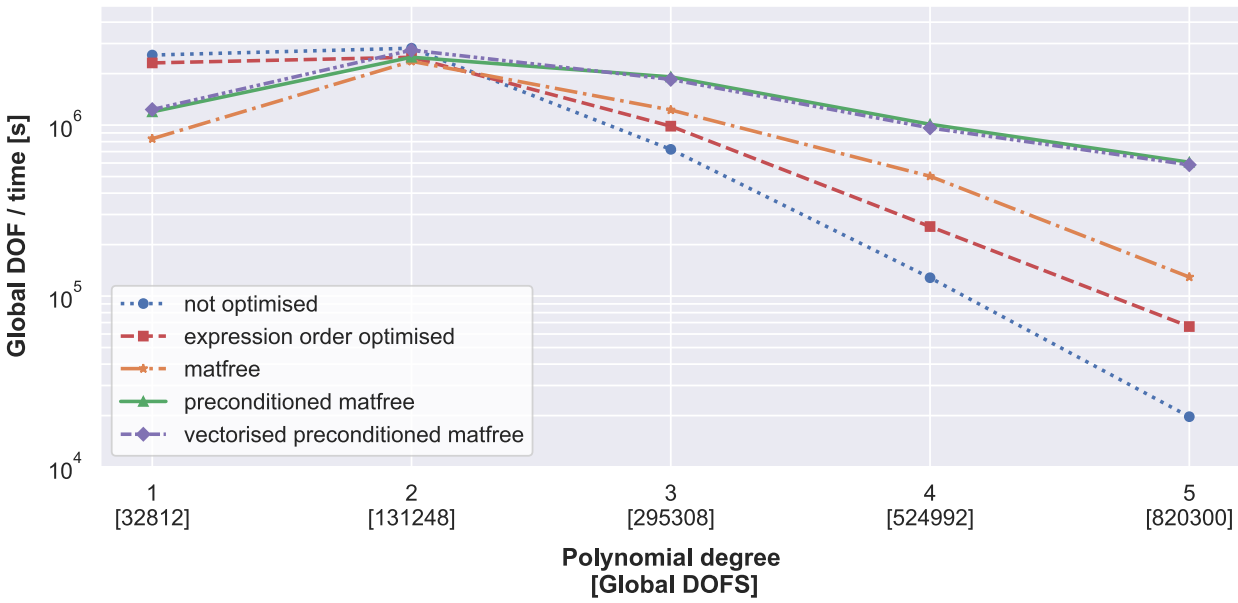


Figure 4.21: DOF throughput of the trace Schur complement action discretised with RTCF_{p-1}-DG_p and polynomial degrees p on $16 \times 16 \times 16$ hexahedral mesh with unit step size assmbled with different optimisation modes previously presented in this chapter

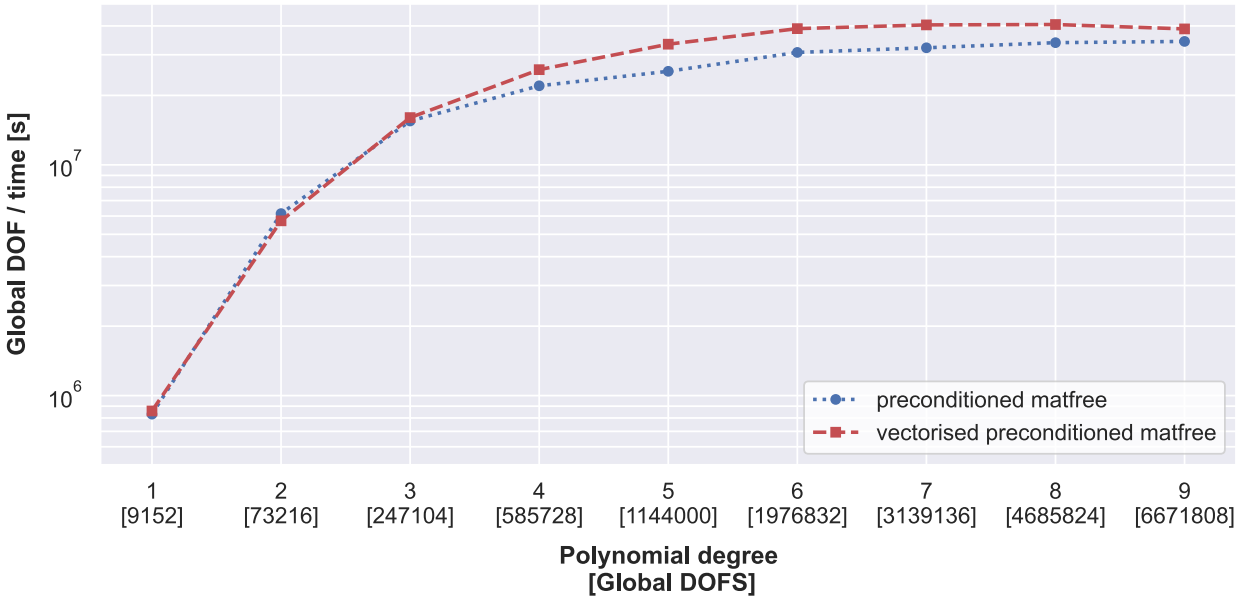


Figure 4.22: DOF throughput of the pressure Schur complement action discretised with RTCF_{p-1}-DG_p and polynomial degrees p on a $16 \times 16 \times 16$ hexahedral mesh with unit step size assembled with the locally matrix-free optimisation modes for higher approximation degrees

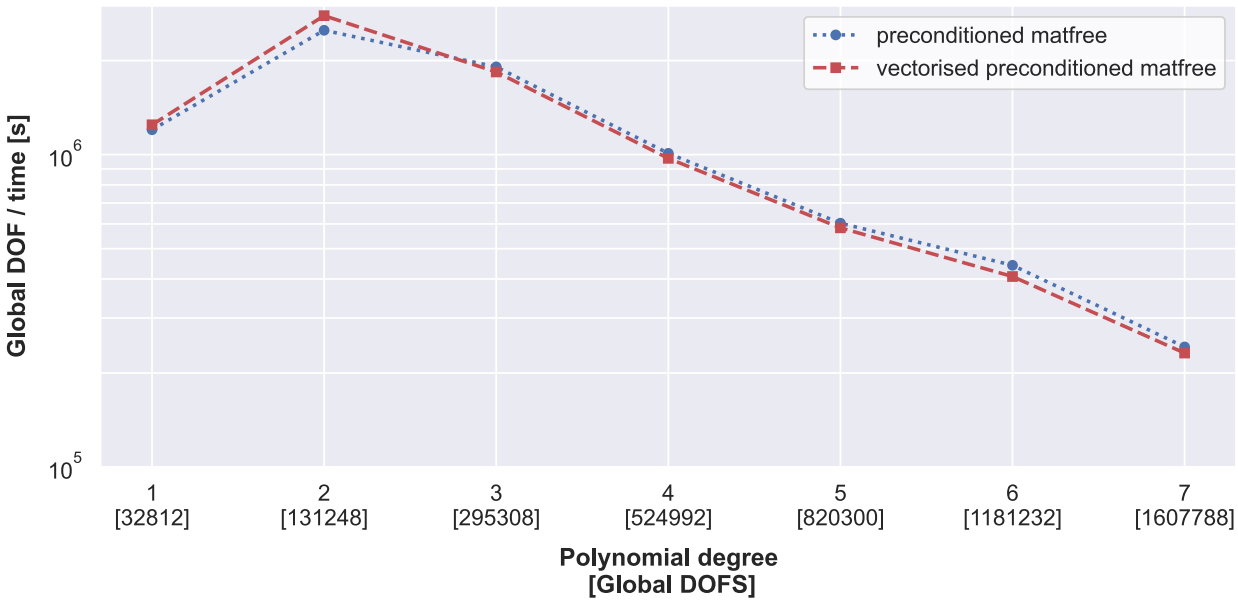


Figure 4.23: DOF throughput of the trace Schur complement action discretised with RTCF_{p-1}-DG_p and polynomial degrees p on a $16 \times 16 \times 16$ hexahedral mesh with unit step size assembled for the locally matrix-free optimisation modes for high approximation degrees

Chapter 5

Performance evaluation of fully matrix-free hybridisation of a mixed Poisson problem

While a subset of results has already been presented for building blocks of the model problem in previous chapters, those results are extended to more realistic solver setups here. As before, a mixed Poisson problem is considered as the model problem. In section 4.3.3, the DOF throughput rates for the model problem are compared for different optimisation modes for the action of both the pressure, defined in equation (4.10), and the trace Schur complement operator, defined in equation (4.12). These operators are building blocks to a fully matrix-free hybridisation preconditioner, as explained in section 4.1. Good throughput, and in particular scaling, has already been demonstrated for both operator actions in the fully matrix-free mode in the previous chapter. The trace Schur complement action throughput in the fully matrix-free mode was shown to slightly trail off with increasing polynomial degree, however, due to a lack of a polynomial degree robust, local preconditioner. Instead of only timing building blocks the performance is now investigated in a full hybridisation solver setup also including accuracy and size information.

In this section, only the expression-order optimised but matrix-explicit, and the matrix-free optimisation mode are considered. A mixed Poisson problem is solved with different solver setups, in most cases with a hybridisation preconditioner, but also with a PETSc fieldsplit preconditioner for a reference. Further, Gopalakrishnan-Tan multigrid (GTMG), which is expected to be a robust preconditioner for the trace system solve with respect to an increasing approximation degree, is investigated. An introduction to GTMG can be found in section 2.2.4.

First, the different solver setups are compared, evaluating both runtime and the iteration count required for a converged solution, see section 5.3. Further, the solvers are evaluated and compared with help of the Time-Size-Accuracy (TAS) spectrum [81], see section 5.4. The spectrum is useful to compare different algorithms and discretisations considering accuracy and throughput rate on top of time-to-solution.

5.1 Problem

The problem of interest is a mixed Poisson problem. The definition of a mixed Poisson problem and how it can be solved with help of hybridisation is explained in the introduction in section 2.2.3. The details on the numerics of the problem are explained in section 4.1.2. In this chapter the Poisson problem is solved on an $L \times L \times L$ domain with $L = 2$ and with a $16 \times 16 \times 16$ mesh for all solvers. The exact solution $u(x, y, z) = x \cdot (L - x) \cdot y \cdot (L - y) \cdot z \cdot (L - z) \cdot e^{L-x+1}$ with L, the characteristic length of the domain, is known through the Method of manufactured solutions (MMS) [82].

All runs are verified in the actions of the newly introduced performance testing framework in [77]. Data, plots and script are published in the same repository. The Firedrake version required is the same as in section 4.3.1.

5.2 Solvers

5.2.1 General setup

The solver setups are constructed in a hierarchical way, where parts of a solver are swapped for what would be expected to be a better choice, one after the other. All cases involve a hybridisation preconditioner, besides the first one - the native DG solver showcases the performance of a PETSc fieldsplit preconditioner as a reference.

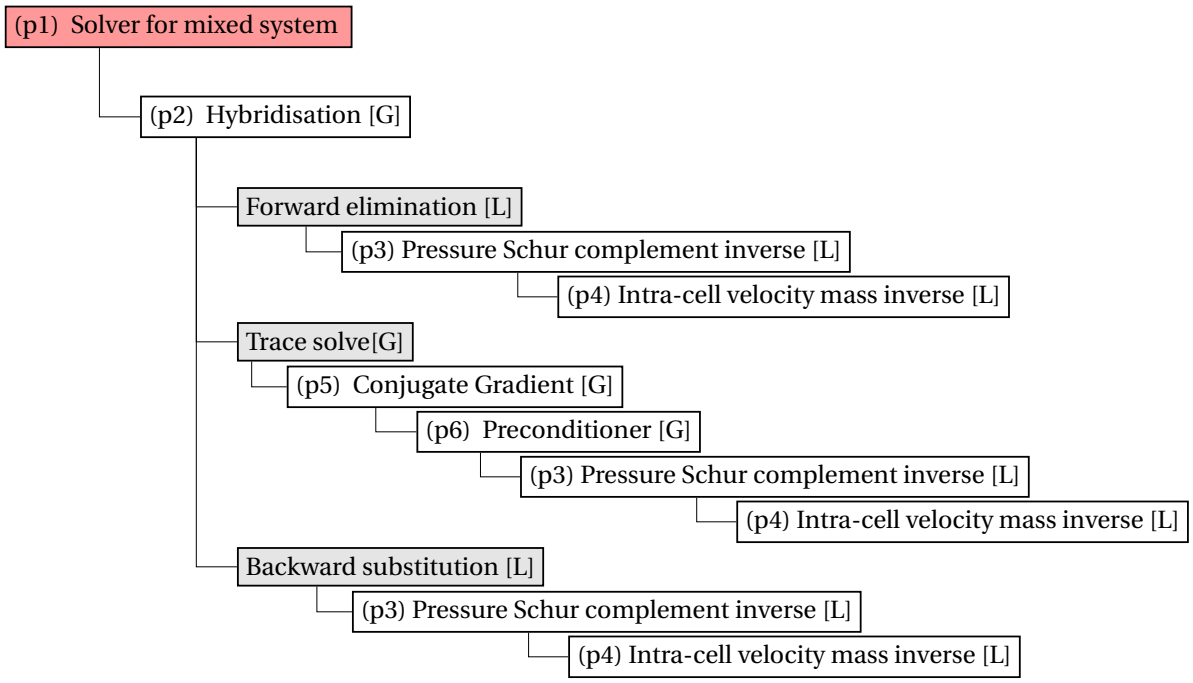


Figure 5.1: General diagram of the Hybridisation solver setup. The forward elimination and backward substitution are purely local kernels (denoted by [L]), whereas the trace solve involves global kernels (denoted by [G]) and local ones. Red denotes the outermost solve, gray separates parts of the hybridisation preconditioner.

From case 2 to case 7, the global solver on the trace system of the hybridisation preconditioner (for equation (4.2)) is changed from a direct LU solver, to an iterative CG solver, and further, from a matrix-explicit to a globally matrix-free to a fully matrix-free solver. The preconditioner on the trace system solve is swapped from Jacobi to assembled Jacobi to GTMG in the hierarchy of the cases. In addition, for the fully matrix-free setup the local solvers are considered in an unpreconditioned and preconditioned version. For clarity, all cases are listed in table 5.1 with explanation and links to their corresponding solver diagrams in the appendix. The solver diagrams involving the hybridisation preconditioner (case 2 - 7) follow the structure in diagram 5.1. Which solvers are chosen for the trace preconditioner (node (p6)) and the local solves (nodes (p3) and (p4)) depends on the case and details are explained in section 5.2.2 and 5.2.3.

	Solver setup	Details	Fig.
1	Petsc's fieldsplit	Reference solver	9.6
2	Hybridisation + matrix-explicit CG + Jacobi + local, direct solves	Hybridisation with a matrix-explicit, iterative CG solver preconditioned with Jacobi on the trace system and direct solves for the local systems	9.7
3	Hybridisation + matrix-explicit CG + GTMG + local, direct solve	Same as case 2, but with a GTMG preconditioner on the trace system	9.7
4	Hybridisation + globally matfree CG + Jacobi + local, direct solve	Same as case 2, but with a globally matrix-free CG solver preconditioned with an assembled Jacobi preconditioner on the trace system	9.7
5	Hybridisation + globally matfree CG + GTMG + local, direct solve	A combination of the trace solver from case 4 and trace preconditioner from case 3: a globally matrix-free CG solver preconditioned with the GTMG preconditioner for the trace system	9.7
6	Hybridisation + fully matrix-free CG + GTMG + locally unpreconditioned CG	Same as case 5, but with a fully matrix-free global CG solver for the trace system and unpreconditioned, local CG solvers	9.8
7	Hybridisation + fully matrix-free CG + GTMG + locally preconditioned CG	Same as case 7, but with preconditioned, local CG solvers as explained in section 4.2.4	9.9

Table 5.1: A summary of the solver setups. Case 1 is a reference solver from PETSc without using Firedrake's preconditioning, the others are using the Hybridisation preconditioner. The Hybridisation preconditioner is the Firedrake equivalent of a PETSc fieldsplit preconditioner with the first split for the intra-cell velocity and pressure contributions and the second fieldsplit for the velocity trace contributions, but using local, solves for intra-cell velocity and pressure systems and a global solve for the trace system.

In a later analysis, some test cases are considered in groups to simplify the performance evaluation. The cases are grouped by the type of matrices used in the corresponding solver. The groups are listed in the following.

- **Group 1**: All cases which are *fully matrix-explicit*, which are number 1 to 3.
- **Group 2**: All cases which are *globally matrix-free*, which a are number 4, 5.
- **Group 3**: All cases which are *fully matrix-free*, which are number 6, 7.

5.2.2 Trace preconditioner

In the all cases besides 1, 2 and 4, GTMG preconditions the trace solve (node (p6) in diagram 5.1). Inside the GTMG preconditioner there are two different multigrid algorithms. The GTMG p-multigrid solves a fine space problem and coarse space problem. On the coarse space problem, an h-multgrid is used with a fine mesh solve and a coarse mesh solve. A diagram which zooms in onto node (p6) in diagram 5.1 for a GTMG preconditioner is presented in figure 5.2.

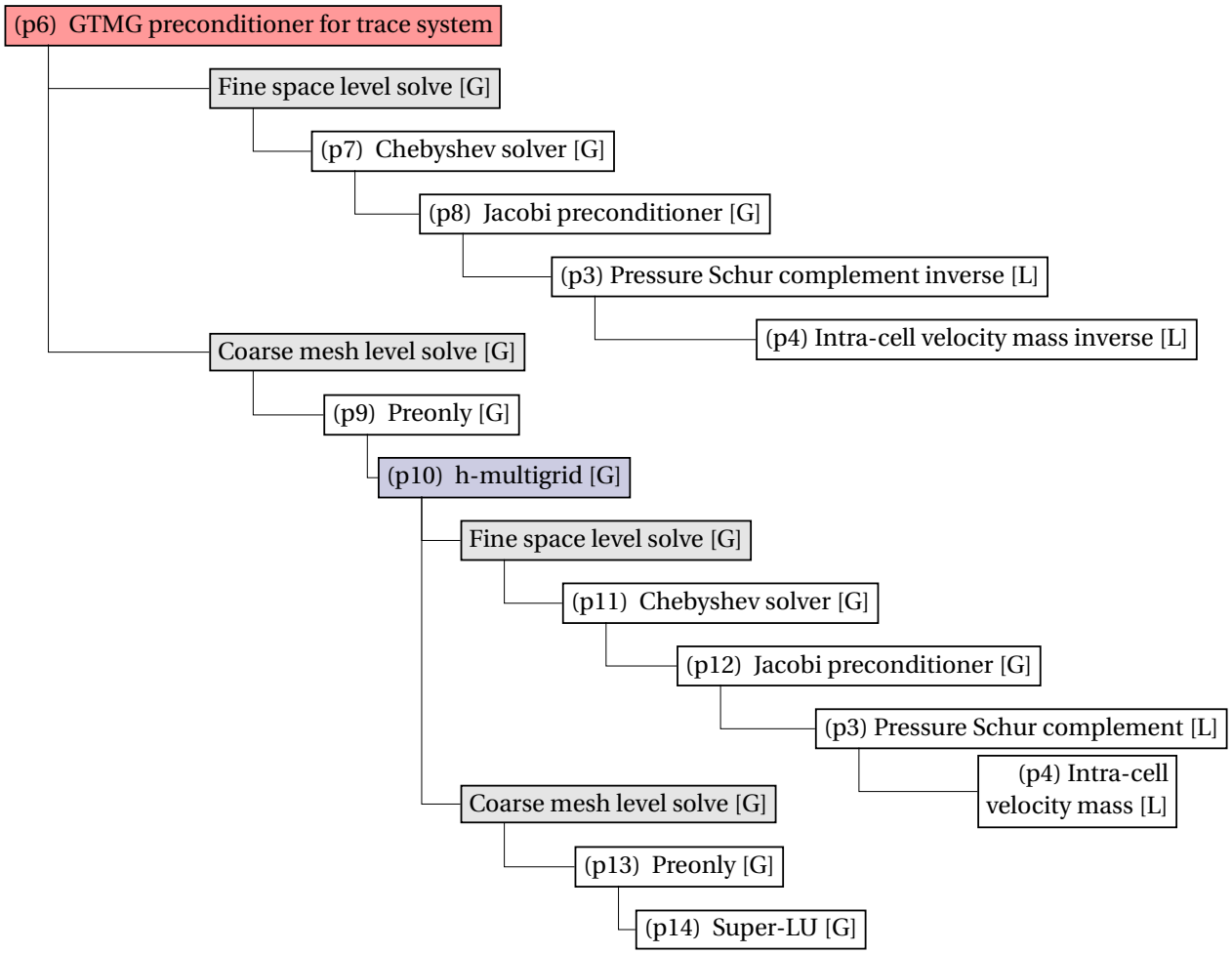


Figure 5.2: Setup of the GTMG preconditioner. The diagram is zooming into the solver option specified in node (p6) in diagram 5.1

The preconditioner presented in diagram 5.2 differs slightly between the groups of cases depending on the matrix types involved. For group 1, which contains all fully-matrix explicit cases, the whole GTMG preconditioner setup is matrix-explicit. For group 2 and group 3 , the Jacobi preconditioner on the fine space level solve (node (p8) in diagram 5.2) is assembled explicitly and the coarse mesh level preconditioner is matrix-explicit. In both cases 2 and 4, a matrix-explicit Jacobi preconditioner is used as a trace preconditioner.

5.2.3 Local Solver

Similar to the GTMG preconditioner the local solvers depends on the type of matrices used and therefore, differ between the groups.

For group 1 and group 2 the local solvers (nodes (p3) and (p4) in all solver diagrams) are direct solves. For group 3 the solves are locally matrix-free, iterative solves, in particular the matrix-free Conjugate Gradient method, and depending on the case the local solvers are unpreconditioned (case 6) or both the intra-velocity mass system and the pressure Schur complement system are preconditioned (case 7). Which local preconditioners are used is explained in section 4.2.4.

5.2.4 Tolerances

In this performance evaluation, the domain of the problem stays unchanged while the amount of elements in the mesh and the approximation polynomial degree changes. All solvers are iterated to an accuracy that is sufficient for convergence on the finest mesh and finest approximation degree. This is the easier setup, but not ideal, because on coarser problems the solvers iterate to an accuracy which is not actually necessary and are therefore not achieving the performance they could. In a physical application it should only be iterated to an accuracy that is required and not more. The interest here lies in a comparison of different solvers in particular on the highest order, however, and a setup which is the same across all solver parameter test cases, is considered as unproblematic.

In the parameter setups, there are multiple solvers nested into each other and which accuracy the outer solvers can achieve depends on the inner solvers. Therefore, convergence of the iterative solver is chosen solely in terms of relative tolerances. The outermost solver is iterated to a relative tolerance of $1e-9$, the inner solvers are chosen in dependence of the outer solver tolerance. The exact solver tolerances can be found in the solver diagrams in the appendix in E.

5.2.5 Parallelism

In this section all simulations are run in serial and without vectorisation. The matrix-explicit implementations have a distinct advantage in sequential runs since they get more bandwidth than on a fully populated node. Therefore, further experiments are needed.

5.2.6 Hardware specification

For a full account of hardware specifications refer to section 3.4.1.

	Intel(R) Xeon(R) CPU E5-2640 v3
Base Frequency	2.6
Theoretical peak performance (double-precision)	665.6 GFLOP/s [1]
Memory bandwidth performance	60 GB/s [2]

Table 5.2: Hardware specification for the dual-socket Intel(R) Xeon(R) CPU E5-2640 v3 architecture used for the experiments

5.3 Performance comparison of the different solvers

To give an overview, first, the set of solvers is considered with regards to runtime and iteration count. Both the iterations of the outer solver (node (p17) in diagram 5.1) and the trace solver (node (p5) in diagram 5.1) are taken into account.

All numbers are presented in heatmaps, see figures 5.3 and 5.4. In both heatmaps, performance numbers are shown for all solver test cases and various approximation degrees, but the amount of cells in the mesh are fixed as explained in section 5.1. For the cases where GTMG preconditions the trace solve (node (p6) in diagram 5.1), the coarse (space) solve of the GTMG preconditioner (node (p10) in diagram 5.2) is solved with an h-multigrid, and the coarse mesh level solve of that (node (p15) in diagram 5.2) is solved on a $4 \times 4 \times 4$ mesh. Heatmaps generated for a smaller mesh, an $8 \times 8 \times 8$ mesh (and $2 \times 2 \times 2$ mesh on the coarse mesh solve), can be found in the appendix F.

In Group 1, the group of fully matrix-explicit cases, the reference solver is performing the worst in terms of the time and iteration count of the outermost solver for higher order. For lower order ($p = 0, 1$) the reference solver is highly competitive, but with increasing polynomial degree the number of solver iterations grows faster for the reference solver than for the other solver setups in group 1. The reference solver in case 1 has no iterations in the trace iteration heatmap, because inter and intra-cell velocity are solved at the same time. There is no hybridisation preconditioner used in this case. In contrast, all hybridisation preconditioners are run to convergence so that the outer solver converges in one iteration. Therefore, the outer iterations of case 1 are compared to the trace iterations of case 2 and 3.

[1] Calculated as base frequency × cores × doubles per simd vector × fma × ports as proposed in [28]

[2] Maximum bandwidth achieved on this architecture by STREAM triad benchmark on a variety of cores. For details on the measurement see section B.

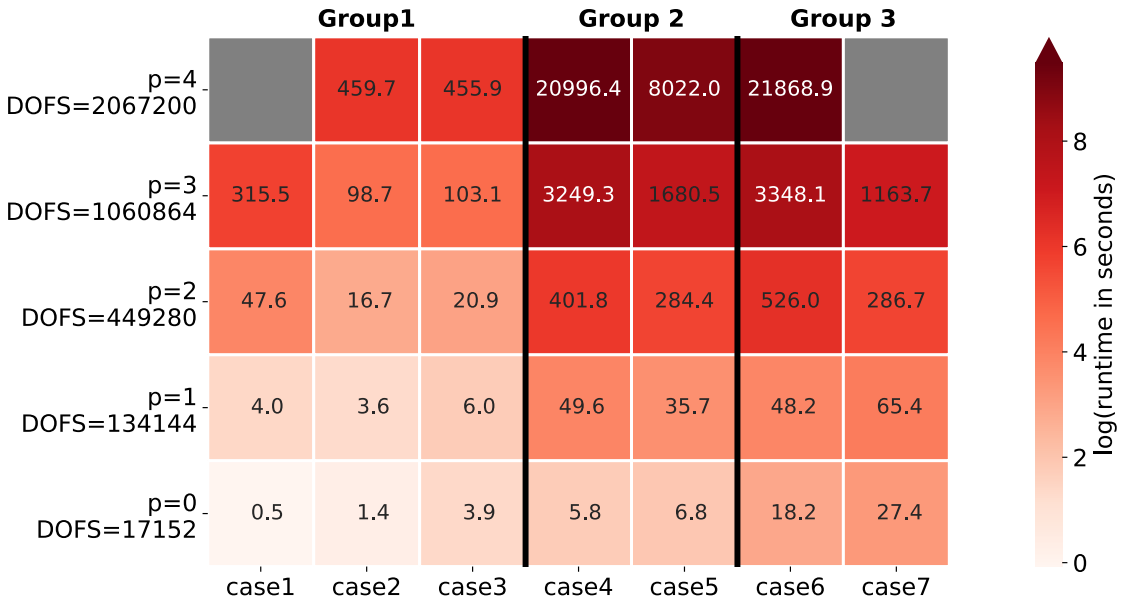


Figure 5.3: Heatmap of runtime performance measurements in seconds for solving the mixed Poisson problem on a fixed mesh (refer to section 5.1). The cases for different solver setups can be found in table 5.1. Gray, unannotated cells correspond to a solver which is not able to solve the problem.

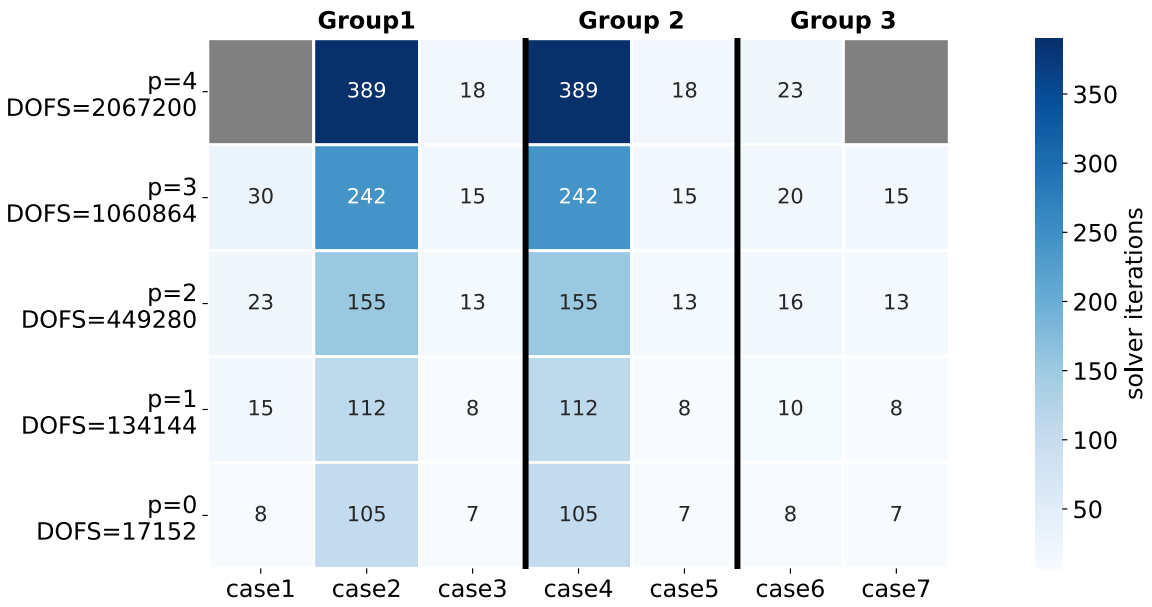


Figure 5.4: Heatmap of outer solver iterations (for case 1) and trace solver iterations (for the other cases) for solving the mixed Poisson problem on a fixed mesh (refer to section 5.1). In case 1, the reference solver, is not using hybridisation and has therefore no trace iterations, but outer iterations. The mapping from the cases to the different solver setups can be found in table 5.1. Gray, unannotated cells correspond to a solver which is not able to solve the problem.

Both, hybridisation with a Jacobi preconditioned (case 2) and a GTMG preconditioned CG (case 3), perform better in terms of the runtime compared to the reference solver. This is not surprising and an argument for using the hybridisation preconditioner as suggested in [21]. In fact, for the highest order approximation the reference solver in case 1 ran out of resources before a solution could be found. The Jacobi preconditioned CG has a strongly increasing iteration number with approximation degree, which is the reason why the GTMG preconditioner is considered instead. Even though the GTMG preconditioner iteration count grows much less with approximation degree, the runtime of the full solver is approximately equal to the one where Jacobi preconditioned CG is used on the trace solve. Possibly, there is a cross-over point at high order, where GTMG outperforms Jacobi.

In group 2, the group of the globally matrix-free solvers, the solver where Jacobi is used (case 4) has a much stronger increase in iteration count with approximation degree for the trace solve, compared to the case where the GTMG preconditioner is used. This is an equivalent observation to the one where case 2 and case 3 are compared. The difference to group 1 is, that the runtime of case 4 is much worse than the runtime of case 5. In addition, it can be noticed that the runtimes in this groups are in general many multiples higher than the ones in group 1.

The problem in group 2 is twofold. First, there is some effort put into setting up the global matrix-free infrastructure (and many global actions are calculated instead of storing the global matrix once), but the local matrices are still matrix-explicit and solving the local systems with a direct solve is a dominating cost in the full solver. Second, currently there is no way to assemble a diagonal (point-diagonal, not block-diagonal) of the operator on the trace solve, see equation (2.35), in a matrix-free way. That is because the expression which represents the diagonal of the trace Schur complement contains complex linear algebra operations and cannot be transformed such that the diagonal acts on a terminal tensor. Therefore, the diagonal of the trace operator has to be assembled in a matrix-explicit manner which is expected to come at a similar cost to assembling the operator itself. A quantification of the cost for the assembly of the diagonal in case 7 can be found in figure 5.11. This is a big problem, which is also encountered in group 3. Both of these issue justify why case 5 performs much worse than case 3. Possibly computing and storing the diagonal upfront and reusing it would resolve some of this runtime bottleneck. Further, one could consider using a block-Jacobi methods on the smoother [23].

Case 6 in group 3 is the first case where a fully matrix-free solver setup is considered, but the local solves are unpreconditioned. The runtime of this case is worse than its globally matrix-free equivalent case 5. The iterations count at the highest order is also about 20 percent higher which might be the reason for the unexpected increase in runtime from case 5 to case 6. For case 7, which is using preconditioned local matrix-free infrastructure, the solver ran out of resources before it could find a solution. One of the reasons for that is likely the lack of a matrix-free diagonal preconditioner on the matrix-free, fine space levels of the GTMG preconditioner.

Further, while the fully matrix-free application of the trace Schur complement operator, the operator of the smoother in the GTMG algorithm, showed better throughput than the matrix-explicit version in the previous chapter, many more operations are required and in terms of runtime, the matrix-free application may well perform worse. Further, the matrix-free solver for the trace Schur complement is not robust with respect to the polynomial degree.

The reason for why assembling the diagonal of the trace Schur complement in a matrix-explicit way breaks the solver in case 7, but not in 6, is that the local expressions become more complex as soon as local preconditioners are introduced and therefore more storage is required for the assembly. For the next-to-highest order runs the preconditioned fully-matrix hybridisation is outperforming both the global-matrix free and the unpreconditioned fully-matrix-free cases (case 5 and 6), but not the fully matrix-explicit version.

In summary, as expected matrix-explicit Jacobi preconditioned hybridisation outperforms the reference solver and matrix-explicit GTMG preconditioned hybridisation outperforms Jacobi preconditioned hybridisation. Switching to either globally matrix-free or fully matrix-free GTMG preconditioned hybridisation does not currently pay off, even on higher order approximations, due to a lack of a matrix-free preconditioner on the levels of the fine space solve on the trace system. Further, the matrix-explicit implementations have a distinct advantage in sequential runs since they get more bandwidth than on a fully populated node. More work is needed.

5.4 The Time-Accuracy-Size spectrum

In the following, accuracy and size are taken into account in the investigation, in addition to the runtime. The Time-Accuracy-Size (TAS) spectrum, developed in [81], is useful for comparing the performance of different numerical discretisations and the solvers for their corresponding equation systems simultaneously. This is made possible by linking the traditionally presented performance in form of convergence and scaling analysis in two new plots evaluating the so-called efficacy and true static scaling of the algorithm.

Measure	**Acronym**	**Definition**
Degrees of Freedom	DoFs	number of unknowns
Error	err	$\|\|u_h - u\|\|_{L_2}$
Digits of Accuracy	DoA	$-\log_{10}(\text{err})$
Digits of Size	DoS	$\log_{10}(\text{DoFs})$
Digits of Efficacy	DoE	$-\log_{10}(\text{err} \cdot \text{time})$
Throughput rate	DoFs/s	DoFs/time
True throughput rate	True DoFs/s	DoFs/time $\cdot$ (DoA/DoS)

Table 5.3: Measures for the TAS spectrum with an approximation u_h and an exact solution u

In table 5.3, the measures involved in the TAS spectrum are introduced. Further, the plots in the TAS spectrum, their definition and what question they answer, can be found in table 5.4. Details on the interpretation of the TAS spectrum can be found in [81]. The spectrum is designed such that having a higher value on the y-axis means a better performance in all plots.

A difference compared to the original publication [81] is that here, the TAS spectrum is adapted to mixed FEM. The errors of velocity and pressure are considered separately, while the time-to-solution for each is the time for the mixed solution. Further, while it is later shown that the work scales linearly with the mesh size, this is not true when methods on approximations of different orders are compared. Therefore, the slope of the efficacy does not match the prediction in [81].

Plot	**Definition**	**Answers the Question**
Convergence	DoA versus DoS	How much accuracy can be achieved given a discretisation of a fixed size and at what rate?
Static scaling	Throughput rate versus time	At what rate is a solution computed at a given time-to-solution?
Efficacy	DoE versus Time	How well does the spent time translate into the accuracy of the solution?
True static scaling	True throughput rate versus time	How fast is the algorithm if the DoFs are scaled by the accuracy they contribute?

Table 5.4: Plots in the TAS spectrum

5.4.1 Comparison across approximation degree and mesh sizes

In figure 5.5 and 5.6, the TAS spectrum is presented for velocity and pressure for case 3, the case of the best performing fully matrix-explicit solver. The increase in size for a fixed polynomial approximation degree (per line) is achieved by refining the mesh.

As expected mesh convergence is achieved. The finer the mesh, the higher is the accuracy achieved, across all polynomial approximation degrees. The slope of the different lines scale with the convergence rate α, defined through err $<= Ch^{\alpha}$, where C is a constant and α is dependent on the polynomial degree p. The optimal convergence rate is stated as $\alpha = p + 1$ [83], [84]. The highest amount of accuracy is achieved in the approximation of the pressure on an RTCF5-DG4 discretisation on the finest mesh. This accuracy is dictating the solver tolerance required. For case 3 for the finest run the solver tolerance was not low enough. As previously mentioned , the solvers across all orders iterate to the same tolerance, see section 5.2.4.

Evaluating the static scaling plot, a constant line is expected. The trailing off on the left hand side is indicating that the solver is acting in a bandwidth limited regime. The static scaling is showing a lower performance for higher order approximation due to a lower throughput rate, even if it is scaled by the accuracy as shown in the true static scaling plot. This matches the expectations since the solver is fully matrix-explicit and loading the data becomes a bottleneck in the algorithm.

While the throughput rates indicate a lower performance of higher order approximations, the efficacy increases with approximation degree. While the time-to-solution increases with increasing approximation degree, if the same time is spent, a high order approximation will yield better accuracy than a lower order approximation on a problem of the same size.

All these results are equally reflected in the TAS spectrum for case 7. The only difference is that the reason for the decreasing throughput rate with polynomial degree is due to the lack of a fully matrix-free preconditioner for the smoother on the coarse space level of the GTMG algorithm and the lack of a robust local preconditioner for the pressure Schur complement. This means that the work does not scale linearly with the approximation degree. Further investigation into the performance per polynomial degree on a fixed mesh is presented in the following section.

5.4.2 Comparison across solver algorithms and approximation degrees

In figure 5.9 and 5.8, a TAS spectrum is shown considering different solver algorithms. Each line represents one of the previously listed test cases. The focus is set on case 3, 5, 6 and 7. Each point in the lines presents the performance on a different approximation. While the mesh is fixed to $4 \times 4 \times 4$ on the coarse mesh, and $16 \times 16 \times 16$ on the fine mesh, the polynomial degree p in an RTCF-$(p+1)$ and DG-(p) discretisation ranges from 0 to 4.

Given that setup, the accuracy rate scales with polynomial approximation degree and is exponential, as expected. The DoA is the same for each test case because the solvers are iterated to convergence.

All test cases show a trailing off in the throughput rate with increasing polynomial degree. For the fully matrix-free cases, this is due to the lack of both a matrix-free preconditioner on a global level and the lack of an approximation degree independent preconditioner on the local pressure Schur complement solves. The lack of p-robustness of the local pressure Schur complement solves is e.g. shown in table 4.1. A better local preconditoner has to be found. If the global assembly of the diagonal trace Schur complement operator would not exceed the storage requirements, case 7 would possibly beat the global matrix-free case, case 5. The local preconditioning in case 7 improves the runtime and equally the throughput rates, compared to the locally unpreconditioned case, case 6.

Both the true static scaling and the efficacy plots show a superiority of the matrix-explicit, GTMG preconditioned hybridisation compared to its globally and fully matrix-free counterparts. More work on finding good preconditioners is required. It has to be mentioned that the TAS spectrum does not account for the arithmetic intensity of the algorithm. There are many more FLOPS processed in fully matrix-free methods.

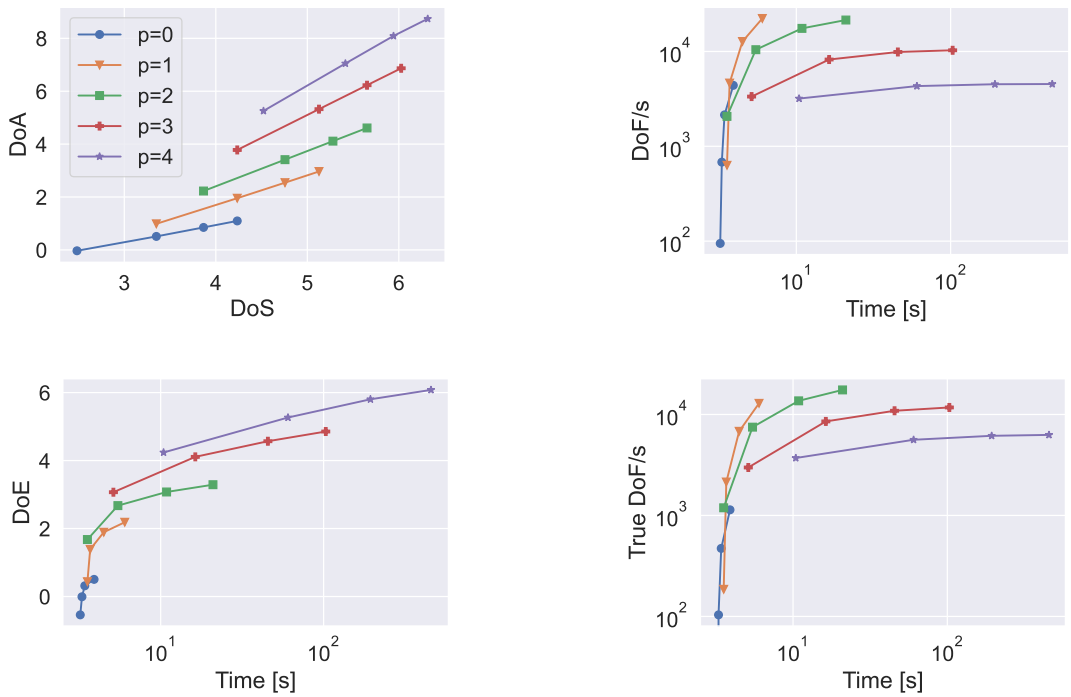


Figure 5.5: *Case 3*: Full TAS spectrum for *velocity*: mesh convergence, static scaling, efficacy, true static scaling (left to right, top to bottom). The increase in size between the points on each line is achieved by refining the mesh as follows: $4 \times 4 \times 4$, $8 \times 8 \times 8$, $12 \times 12 \times 12$, $16 \times 16 \times 16$.

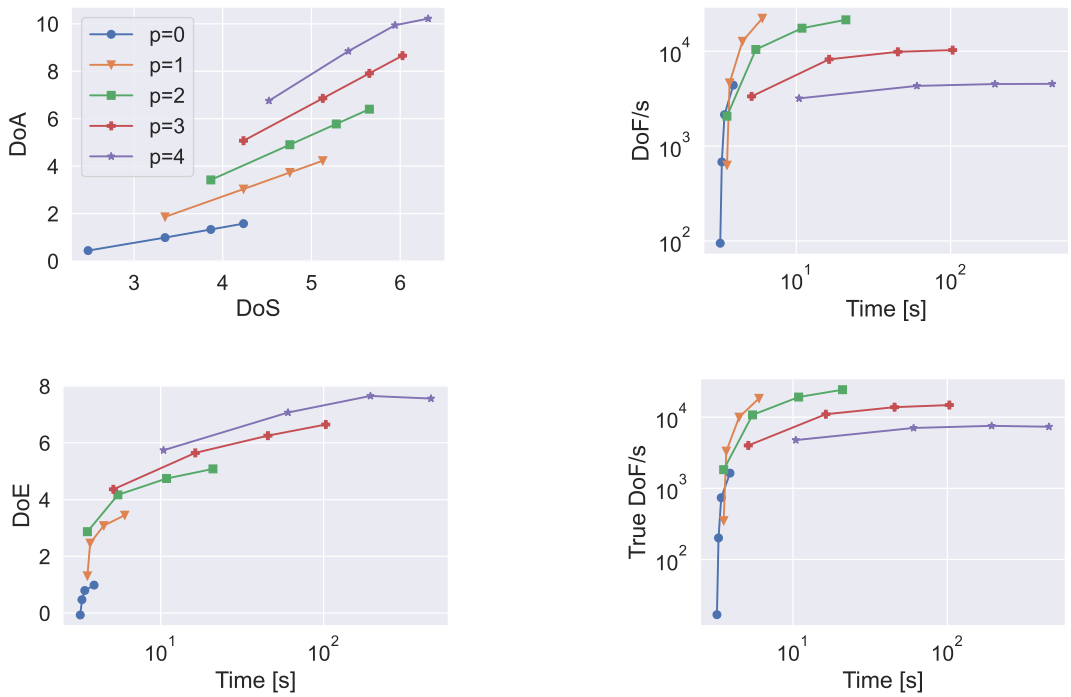


Figure 5.6: *Case 3*: Full TAS spectrum for *pressure*: mesh convergence, static scaling, efficacy, true static scaling (left to right, top to bottom). The increase in size between the points on each line is achieved by refining the mesh as follows: $4 \times 4 \times 4$, $8 \times 8 \times 8$, $12 \times 12 \times 12$, $16 \times 16 \times 16$.

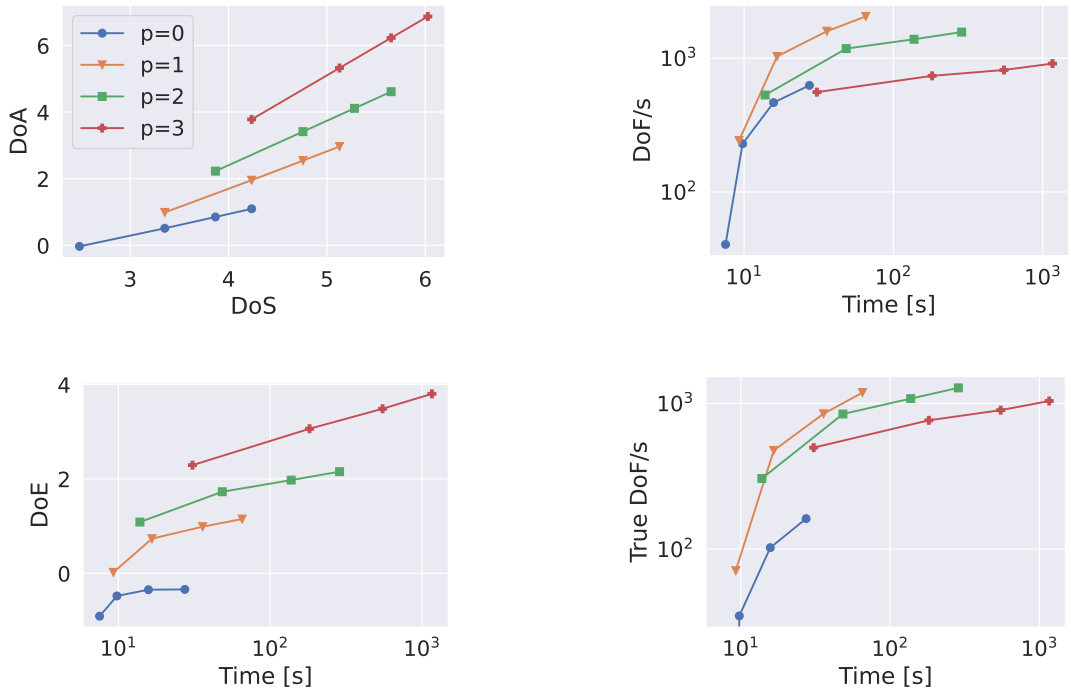


Figure 5.7: *Case 7*: Full TAS spectrum for *velocity*: mesh convergence, static scaling, efficacy, true static scaling (left to right, top to bottom). The increase in size between the points on each line is achieved by refining the mesh as follows: $4 \times 4 \times 4$, $8 \times 8 \times 8$, $12 \times 12 \times 12$, $16 \times 16 \times 16$.

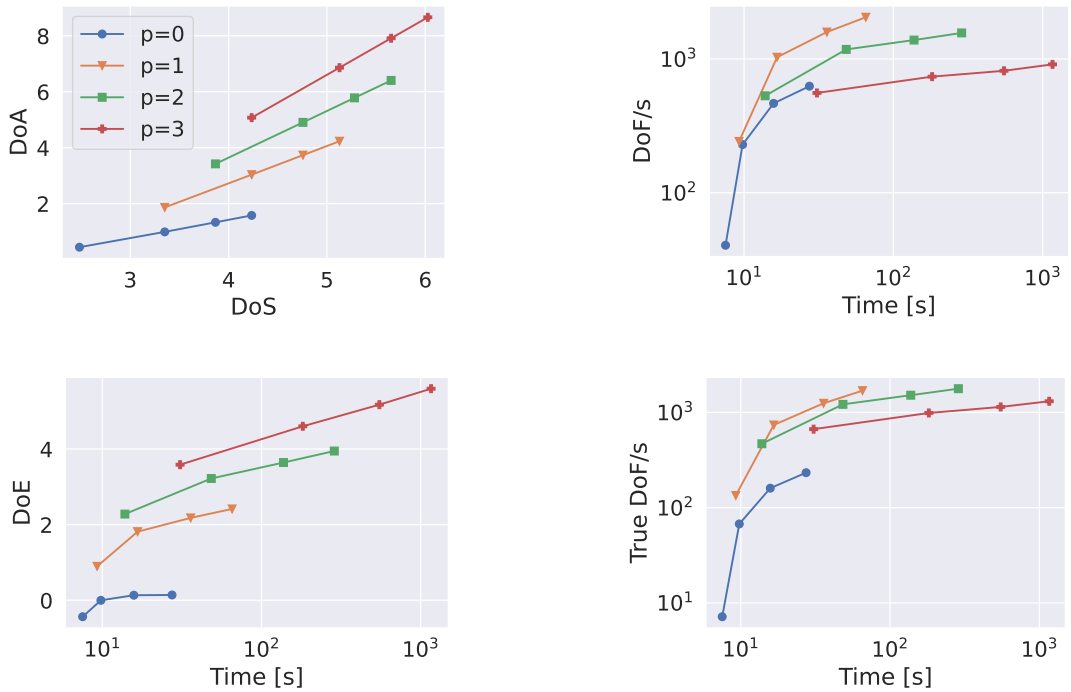


Figure 5.8: *Case 7*: Full TAS spectrum for *pressure*: mesh convergence, static scaling, efficacy, true static scaling (left to right, top to bottom). The increase in size between the points on each line is achieved by refining the mesh as follows: $4 \times 4 \times 4$, $8 \times 8 \times 8$, $12 \times 12 \times 12$, $16 \times 16 \times 16$.

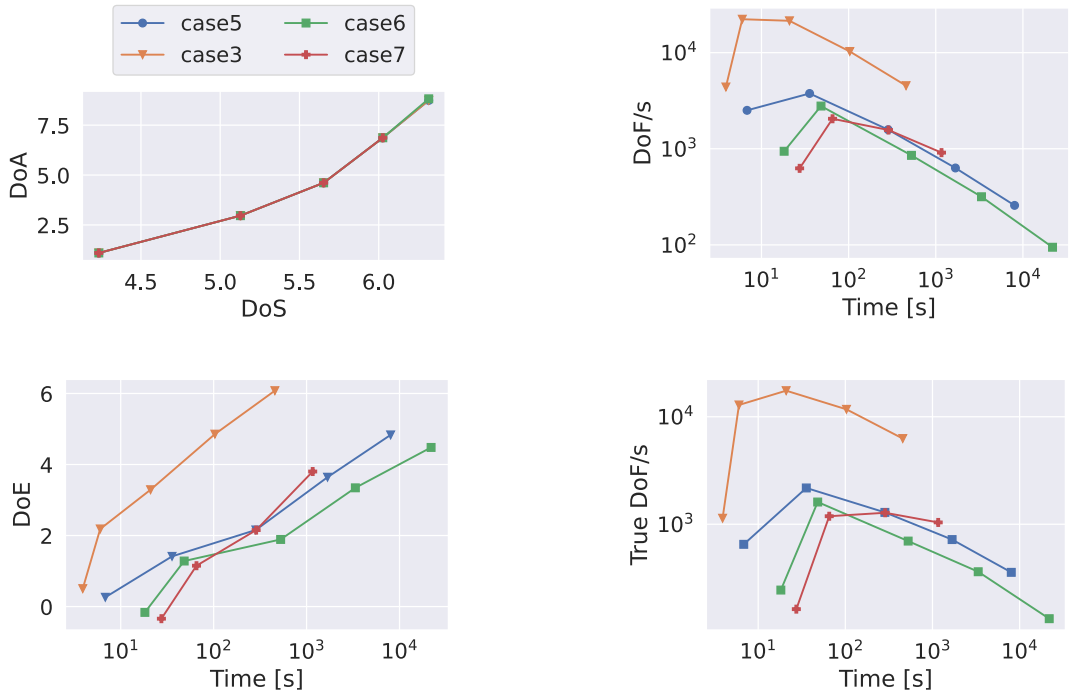


Figure 5.9: Full TAS spectrum for *velocity* for all cases on a $16 \times 16 \times 16$ mesh: mesh convergence, static scaling, efficacy, true static scaling (left to right, top to bottom). The increase in size between the points on each line is achieved by refining the mesh by increasing the approximation degree from 0 to 4

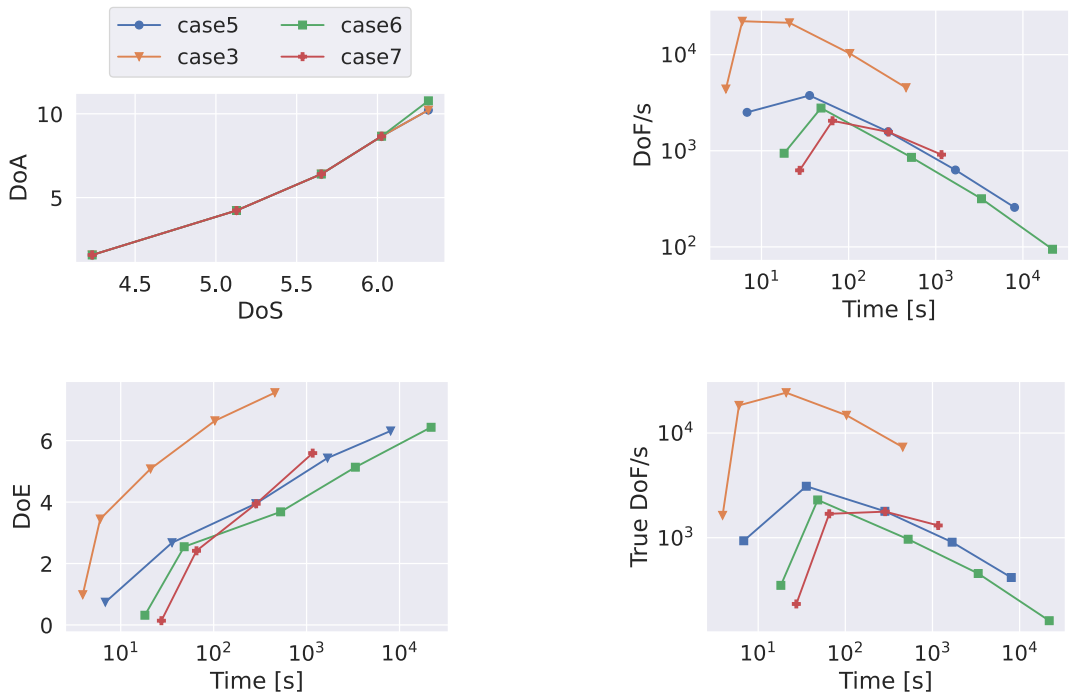


Figure 5.10: Full TAS spectrum for *pressure* for all cases on a $16 \times 16 \times 16$ mesh: mesh convergence, static scaling, efficacy, true static scaling (left to right, top to bottom). The increase in size between the points on each line is achieved by refining the mesh by increasing the approximation degree from 0 to 4.

5.5 A performance profile

A runtime profile gives another perspective on the performance of locally preconditioned, fully matrix-free hybridisation (case 7). The exact timings of the profile are neglected in the evaluation here, only the relation between timings of different parts of the solvers are considered. The performance profile is used to clarify that there are two problems left to solve to achieve high performance for fully matrix-free hybridisation. In figure 5.11, it can be seen that while the assembly of the diagonal of the trace Schur complement operator (green) contributes a big part to the time-to-solution and is certainly the storage bottleneck, the local solvers (blue and purple) in the smoother on the coarse space level of the GTMG preconditioner (yellow) are contributing a big part to the time-to-solution, too. A similar result has been observed in the previous chapter, in section 4.3.3 in figure 4.21, the pressure Schur complement solve is not robust with the polynomial degree.

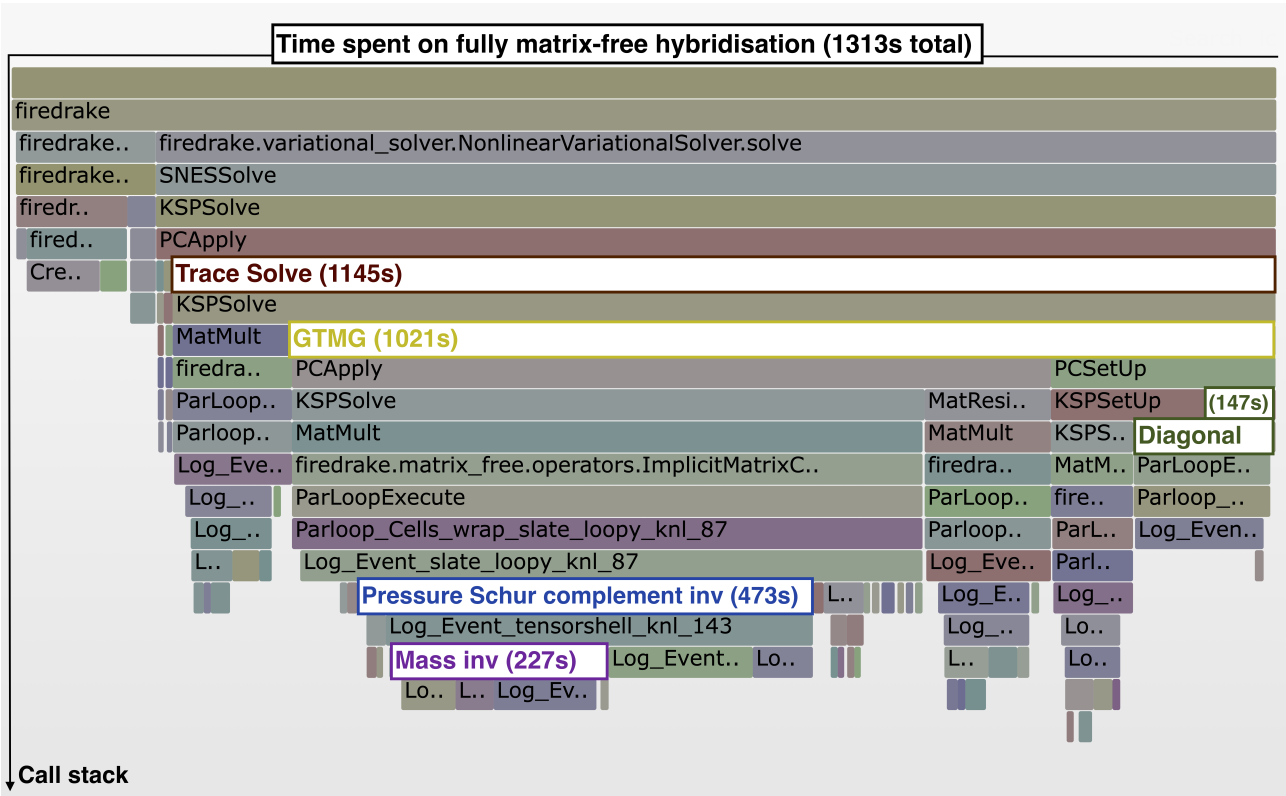


Figure 5.11: A flamegraph of locally preconditioned, fully matrix-free hybridisation (case 7) for a mixed Poisson problem discretised with RTCF4-DG3 on a $16 \times 16 \times 16$ mesh

5.6 On vectorisation

It has been demonstrated in the results section of the previous chapter, section 4.3.3, that it ought to be possible to present some runtime results for vectorised, fully matrix-free hybridisation. But the PyOP2 compilation times of the kernels in the trace solve is so high that one needs to investigate a way to improve the compilation before.

While it takes 71 percent of the run of the lowest order in case 6 for no vectorisation, it takes 95 percent for vectorisation. Instead of 280 seconds, 2400 seconds are spent in total on the compilation. The time in the compilation is mostly spent in loopy functions. The problem is that all local kernels in the global kernel need to be inlined into each other by Loo.py before vectorising, which takes a lot of time in itself. Cross function call vectorisation is currently not supported by Loo.py. Further, the inlining blows up the local kernels and generating the C code from loopy becomes an even bigger bottleneck in the compilation for the vectorised trace solve than for the unvectorised one.

Chapter 6

Summary and future work

In this thesis, the code generation for vectorised, local, matrix-free linear algebra operations on Finite element tensors was developed. The new code infrastructure is used to solve the hybridisation of a mixed Poisson problem fully matrix-free. In chapter 3, vectorised local linear algebra operations have been proven to achieve a high amount of the peak performance and good scaling was shown for fully matrix-free local linear algebra expressions in chapter 4. High performance in a full solver setup as presented in chapter 5 requires more work on the preconditioners to the systems. In the following, I summarise my contributions and how I envisage my work to be continued.

6.1 The Slate compiler and vectorisation

As a stepping stone to my later work on vectorised matrix-free methods, I have rewritten the Slate compiler to compile from Slate to Loo.py rather than to Eigen. Besides a more consistent code generation across the different components of the Firedrake framework, this a) paved the path to the reutilisation of the vectorisation as employed in [28] and b) gives easy access to the code optimisations, in particular sum-factorisation, established in [34]. The new Slate compiler translates the linear algebra operations to GEM in a first stage, and accesses TSFC technology to translate GEM to Loo.py in a second stage.

The inherent structure of finite element methods, in particular the repeated application of the same kernel to the elements of the mesh, makes vectorisation an obvious choice for speeding up the solve of the resulting equation systems. High-order FE methods, as they are considered in this thesis, are data-intensive methods. Only for operations on a reasonable amount of data, the vectorisation overhead through resorting loops diminishes.

A cross-elemental vectorisation strategy has been implemented in Firedrake with help of Loo.py in [28]. The results in the study in [28] are extended to local linear algebra expressions in this thesis. The vectorisation of a simple linear algebra expression has been shown to achieve a high amount of peak performance on a variety of finite element forms. Further, in section 4.3 vectorisation has shown to improve the DOF throughput rate for a fully matrix-free linear algebra expression.

In section 5.6, I pointed out the slow compilation times for Slate kernels. While the wait times are tremendously long for vectorised kernels, they are also unacceptably slow for unvectorised kernels. In the future, it should be investigated if the compilation time can be improved.

Currently, vectorisation is only supported for CPU targets. Extending the support to other targets would be a valuable step to take in the future. In fact, there is ongoing work to introduce general GPU support in Firedrake, see the development branch `https://github.com/OP2/PyOP2/tree/gpu`. Therefore, an extension of vectorisation to GPU targets should become more than feasible soon. Vector extensions in CPUs and the registers in GPUs differ a) in the length of their registers and b) in their memory capacities. Dependent on the target the vectorisation strategy might vary.

6.2 Sum factorisation and fully matrix-free methods

Using matrix-free methods implies a trade of storage for operations. In order to deliver higher performance than matrix-explicit methods, it is crucial to make the calculation of matrix-vector calculations, including an instant assembly of the tensors, faster than loading them from memory and executing the operation. A key factor for performance is the sum factorisation compiler optimisation , which was introduced in TSFC in [30]. This is of particular importance for high order elements. With sum factorisation the work of a matrix-vector product operation reduces from $\mathcal{O}(p^{2d})$ to $\mathcal{O}(p^{d+1})$ [30], where p is the polynomial degree and d is the dimension.

In Firedrake, sum factorisation has only been established for quadrilateral and hexahedral elements. Sum factorisation for simplicial elements is more complicated, since their bases are not straightforwardly decomposed into a tensor product. In [24] it was proven, that there are still possibilities to achieve an efficient matrix-free solver using sum factorisation for simplicial elements, when using the right basis functions. This might give a direction for future extensions of the sum factorisation optimisation in TSFC.

Generally, globally matrix-free methods in Firedrake result from [85]. Introducing matrix-free methods for the local tensors in the linear algebra operations expressed in Slate involved more infrastructural and numerical work.

A change in the compiler strategy, the introduction of new components to Slate and GEM and a new compilation pass from Loo.py to Loo.py to resolve local actions had to be implemented. Some of the new components in Slate and GEM are representing an action, a placeholder for a unassembled tensor, a matrix-free solve and a local preconditioner. Furthermore, a numerical investigation had to be performed to single out a highly-performant local solver for the mixed Poisson problem.

6.3 Performance investigation

After introduction of the infrastructure in Firedrake, the performance of fully matrix-free hybridisation for a mixed Poisson problem discretised with a high order, compatible FEM was investigated as a model problem. Firstly, the investigation should be extended to a parallel setting.

While I have found an acceptable solver and preconditioner for the local equation systems, it is crucial to investigate a new preconditioner for the solver of the global trace system. As expected, the results in section 5.3 show that the Golapakrishnan-Tan multigrid (GTMG) preconditioner is approximately p-robust as long as there is a Jacobi preconditioner on the fine space level solve. An exact Jacobi preconditioner cannot be applied in fully matrix-free form for mathematical reasons, however, and its assembly turns out to introduce a prohibitive storage bottleneck. A better fine space solve preconditioner needs to be found.

Further, a better local preconditioner should be considered for the local pressure Schur complement solve, as explained in section 4.3.3 and 5.3. The pressure Schur complement solve takes too much time with an increasing approximation degree because is not robust with the degree. Alternatively, a block preconditioner, e.g. a Riesz map, could be considered for the solver of the mixed inverse in the trace Schur complement. Local block preconditioners are supported by the infrastructure in this thesis and a Riesz map could be provided with a local `AuxiliaryOperator`, similar to the Laplacian in figure 4.19.

The local matrix-free infrastructure is implemented in an extensible way. The local solvers can be used in a different setting from the one presented in this thesis. A primal Poisson problem can be discretised with a continuous Galerkin method. It can be rewritten into a mixed form by separating interior and exterior contributions, and further solved with static condensation. The method in [86] discretises the mixed operator with basis functions that diagonalise the interior blocks and Firedrake allows fast matrix-free evaluation of the statically-condensed operator with the element proposed in [86]. The local linear algebra expressions in the static condensation can be rewritten into a fully matrix-free form through the work presented in this thesis. With the basis presented in [86], good convergence of the local matrix-free interior solves is expected, because under certain conditions the interior blocks become diagonal.

6.4 An application

While high-order methods generally require a sufficient regularity on the solutions, they have been used with great success to simulate turbulent flow with low regularity. Oscillations can be controlled by limiters. For example, in [87] an high-order incompressible Navier-Stokes solver was successfully applied to a turbulent jet with help of flux reconstruction [88], [89].

Ideally the performance should be tested for the full, incompressible Navier-Stokes equations. As explained in section 4.1, for the time-dependent version of the Navier-Stokes equations Predictor-Evaluation-Corrector-Evaluation (PECE) schemes are frequently used [76]. In PECE schemes the velocity is calculated first without being constrained to a zero divergence, but afterwards corrected with help of a pressure solution. In order to calculate the pressure for the correction, a Poisson equation has to be solved. Therefore, the work in this thesis provides a stepping stone to better performance of the solvers for the full Navier Stokes equations if solutions to the outstanding problems are found.

For a more realistic application, it would be interesting to perform a direct numerical simulation (DNS) for inhomogeneously density-stratified, shear-generated turbulence similar to [90]. It would be interesting to see if we can simulate the evolution of Kelvin-Helmholtz instability, in a highly performant manner with the methods developed in this thesis. The governing equations are the Boussineq equations. This problem would be very suitable to our purposes as it is in demand of high resolutions [90], which could be achieved through high order FEM. Simulating Kelvin-Helmholtz instabilities can help to characterise the mixing efficiency in the ocean. The mixing efficiency is important, because its variations due to internal wave breaking in the abyssal ocean can have a big impact on the ocean's overturning circulation [91]. The faster the waters in the ocean are circulating, the faster the tracers in the water are transported. Carbondioxide, for example, could be released a lot faster by the ocean into the atmosphere, than is currently predicted with a constant mixing efficiency, which in turn can have a big impact on the world's carbon budget.

Acronyms

BOP Block optimisation pass
CG Conjugate Gradient
COFFEE Compiler For Fast Expression Evaluation
CPU central processor unit
DAG directed acyclic tree
DNS direct numerical simulation
DSL domain specific language
FEEC Finite Element Exterior Calculus
FEM Finite Element Method
FFC FEniCS Form Compiler
FIAT Finite Element Automated Tabulator
FInAT smarter library for finite elements, called Finite Element not Automated Tabulator
GEM Tensor Algebra Language
GMRES Generalized Minimal Residual
GPU graphic processor unit
GTMG Golapakrishnan-Tan multigrid
KSP Krylov Subspace Method
MOP Multiplication optimisation pass
PDE partial differential equation
PECE Predictor-Evaluation-Corrector-Evaluation
PETSc Portable, Extensible Toolkit for Scientific computation
PyOP2 High-level Framework for Performance-portable Simulations on Unstructured Meshes
SIMD single instruction multiple data
Slac Slate Compiler
Slate System for Linear Algebra on Tensor Expressions
TSFC Two-stage Form Compiler
TSSLAC Two Stage Linear Algebra Compiler
UFL Unified Form Language

Bibliography

[1] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. McRae, G.-T. Bercea, G. R. Markall, and P. H. J. Kelly, "Firedrake: Automating the finite element method by composing abstractions", *ACM Trans. Math. Softw.*, vol. 43, no. 3, 24:1–24:27, 2016, ISSN: 0098-3500. DOI: 10.1145/2998441. arXiv: 1501.01809. [Online]. Available: http://arxiv.org/abs/1501.01809.

[2] T. H. Gibson, L. Mitchell, D. A. Ham, and C. J. Cotter, "Slate: Extending firedrake's domain-specific abstraction to hybridized solvers for geoscience and beyond", *Geoscientific model development*, vol. 13, no. 2, pp. 735–761, 2020.

[3] G. Karniadakis and S. Sherwin, *Spectral/hp element methods for computational fluid dynamics*. Oxford University Press, 2013.

[4] P. Fischer, J. Lottes, and H. Tufo, "Nek5000", Argonne National Lab. (ANL), Argonne, IL (US), Tech. Rep., 2007.

[5] C. D. Cantwell, D. Moxey, A. Comerford, A. Bolis, G. Rocco, G. Mengaldo, D. De Grazia, S. Yakovlev, J.-E. Lombard, D. Ekelschot, *et al.*, "Nektar++: An open-source spectral/hp element framework", *Computer physics communications*, vol. 192, pp. 205–219, 2015. DOI: 10.1016/j.cpc.2015.02.008.

[6] P. Bastian, M. Altenbernd, N.-A. Dreier, C. Engwer, J. Fahlke, R. Fritze, M. Geveler, D. Göddeke, O. Iliev, O. Ippisch, *et al.*, "Exa-Dune-flexible pde solvers, numerical methods and applications", in *Software for Exascale Computing-SPPEXA 2016-2019*, Springer, 2020, pp. 225–269. DOI: 10.1007/978-3-030-47956-5_9.

[7] D. Arndt, N. Fehn, G. Kanschat, K. Kormann, M. Kronbichler, P. Munch, W. A. Wall, and J. Witte, "ExaDG: High-order discontinuous Galerkin for the exa-scale", in *Software for Exascale Computing-SPPEXA 2016-2019*, Springer, 2020, pp. 189–224. DOI: 10.1007/978-3-030-47956-5_8.

[8] C. J. Cotter and J. Shipton, "Mixed finite elements for numerical weather prediction", *Journal of Computational Physics*, vol. 231, no. 21, pp. 7076–7091, 2012. DOI: 10.1016/j.jcp.2012.05.020.

[9] A. Staniforth, T. Melvin, and C. Cotter, "Analysis of a mixed finite-element pair proposed for an atmospheric dynamical core", *Quarterly Journal of the Royal Meteorological Society*, vol. 139, no. 674, pp. 1239–1254, 2013. DOI: 10.1002/qj.2028.

[10] C. J. Cotter and J. Thuburn, "A finite element exterior calculus framework for the rotating shallow-water equations", *Journal of Computational Physics*, vol. 257, pp. 1506–1526, 2014. DOI: 10.1016/j.jcp.2013.10.008.

[11] A. T. McRae and C. J. Cotter, "Energy-and enstrophy-conserving schemes for the shallow-water equations, based on mimetic finite elements", *Quarterly Journal of the Royal Meteorological Society*, vol. 140, no. 684, pp. 2223–2234, 2014. DOI: 10.1002/qj.2291.

[12] A. Natale, J. Shipton, and C. J. Cotter, "Compatible finite element spaces for geophysical fluid dynamics", *Dynamics and Statistics of the Climate System*, vol. 1, no. 1, 2016. DOI: 10.1093/climsys/dzw005.

[13] W. Bauer and C. J. Cotter, "Energy–enstrophy conserving compatible finite element schemes for the rotating shallow water equations with slip boundary conditions", *Journal of Computational Physics*, vol. 373, pp. 171–187, 2018.

[14] T. M. Bendall, T. H. Gibson, J. Shipton, C. J. Cotter, and B. Shipway, "A compatible finite-element discretisation for the moist compressible Euler equations", *Quarterly Journal of the Royal Meteorological Society*, vol. 146, no. 732, pp. 3187–3205, 2020. DOI: 10.1002/qj.3841.

[15] D. N. Arnold, R. S. Falk, and R. Winther, "Finite element exterior calculus, homological techniques, and applications", *Acta numerica*, vol. 15, pp. 1–155, 2006. DOI: 10.1017/S0962492906210018.

[16] T. Melvin, T. Benacchio, B. Shipway, N. Wood, J. Thuburn, and C. Cotter, "A mixed finite-element, finite-volume, semi-implicit discretization for atmospheric dynamics: Cartesian geometry", *Quarterly Journal of the Royal Meteorological Society*, 2019.

[17] S. V. Adams, R. W. Ford, M Hambley, J. Hobson, I Kavčič, C. Maynard, T Melvin, E. H. Müller, S Mullerworth, A. Porter, *et al.*, "Lfric: Meeting the challenges of scalability and performance portability in weather and climate models", *Journal of Parallel and Distributed Computing*, 2019.

[18] M. Benzi, G. H. Golub, and J. Liesen, "Numerical solution of saddle point problems", *Acta numerica*, vol. 14, pp. 1–137, 2005. DOI: 10.1017/S0962492904000212.

[19] C. Maynard, T. Melvin, and E. H. Müller, "Multigrid preconditioners for the mixed finite element dynamical core of the lfric atmospheric model", *Quarterly Journal of the Royal Meteorological Society*, vol. 146, no. 733, pp. 3917–3936, 2020. DOI: https://doi.org/10.1002/qj.3880. eprint: https://rmets.onlinelibrary.wiley.com/doi/pdf/10.1002/qj.3880. [Online]. Available: https://rmets.onlinelibrary.wiley.com/doi/abs/10.1002/qj.3880.

[20] B. Cockburn and J. Gopalakrishnan, "A characterization of hybridized mixed methods for second order elliptic problems", *SIAM Journal on Numerical Analysis*, vol. 42, no. 1, pp. 283–301, 2004. DOI: 10.1137/S0036142902417893.

[21] T. H. Gibson, "Hybridizable compatible finite element discretizations for numerical weather prediction: Implementation and analysis", PhD thesis, Imperial College London, 2019. DOI: 10.25560/79431.

[22] S. Müthing, M. Piatkowski, and P. Bastian, "High-performance implementation of matrix-free high-order discontinuous Galerkin methods", *arXiv preprint arXiv:1711.10885*, 2017. DOI: 10.48550/arXiv.1711.10885.

[23] P. Bastian, E. H. Müller, S. Müthing, and M. Piatkowski, "Matrix-free multigrid block- preconditioners for higher order discontinuous Galerkin discretisations", *Journal of Computational Physics*, vol. 394, pp. 417–439, 2019. DOI: 10.1016/j.jcp.2019.06.001.

[24] D. Moxey, R. Amici, and M. Kirby, "Efficient matrix-free high-order finite element evaluation for simplicial elements", *SIAM Journal on Scientific Computing*, vol. 42, no. 3, pp. C97–C123, 2020. DOI: 10.1137/19M1246523.

[25] M. Kronbichler and K. Kormann, "Fast matrix-free evaluation of discontinuous Galerkin finite element operators", *ACM Transactions on Mathematical Software (TOMS)*, vol. 45, no. 3, pp. 1–40, 2019. DOI: 10.1145/3325864.

[26] M. Kronbichler and K. Ljungkvist, "Multigrid for matrix-free high-order finite element computations on graphics processors", *ACM Transactions on Parallel Computing (TOPC)*, vol. 6, no. 1, pp. 1–32, 2019. DOI: 10.1145/3322813.

[27] T. Kolev, P. Fischer, M. Min, J. Dongarra, J. Brown, V. Dobrev, T. Warburton, S. Tomov, M. S. Shephard, A. Abdelfattah, *et al.*, "Efficient exascale discretizations: High-order finite element methods", *The International Journal of High Performance Computing Applications*, vol. 35, no. 6, pp. 527–552, 2021. DOI: 10.1177/1094342021102080.

[28] T. Sun, L. Mitchell, K. Kulkarni, A. Klöckner, D. A. Ham, and P. H. Kelly, "A study of vectorization for matrix-free finite element methods", *The International Journal of High Performance Computing Applications*, vol. 34, no. 6, pp. 629–644, 2020. DOI: 10.1177/1094342020945005.

[29] D. Kempf, R. Heß, S. Müthing, and P. Bastian, "Automatic code generation for high- performance discontinuous galerkin methods on modern architectures", *ACM Transactions on Mathematical Software (TOMS)*, vol. 47, no. 1, pp. 1–31, 2020. DOI: 10.1145/3424144.

[30] M. Homolya, R. C. Kirby, and D. A. Ham, "Exposing and exploiting structure: optimal code generation for high-order finite element methods", 2017. arXiv: 1711.02473 [cs.MS]. [Online]. Available: http://arxiv.org/abs/1711.02473.

[31] W. Ritz, "Uber eine neue methode zur losung gewisser variationsprobleme der mathematischen physik", *Journal fur Mathematik*, vol. 135, s–1, 1909.

[32] P. G. Ciarlet, *The finite element method for elliptic problems*. SIAM, 2002.

[33] M. E. Rognes, R. C. Kirby, and A. Logg, "Efficient assembly of H(div) and H(curl) conforming finite elements", *SIAM Journal on Scientific Computing*, vol. 31, no. 6, pp. 4130–4151, 2010. DOI: 10.1137/08073901X.

[34] M. Homolya, "On code generation techniques for finite elements", PhD thesis, Imperial College London, 2018. DOI: 10.25560/62636.

[35] D. Arnold, R. Falk, and R. Winther, "Finite element exterior calculus: from Hodge theory to numerical stability", *Bulletin of the American mathematical society*, vol. 47, no. 2, pp. 281–354, 2010.

[36] D. N. Arnold, *Finite element exterior calculus*. SIAM, 2018, vol. 93.

[37] G. A. Wimmer, "Energy conserving compatible finite element methods for numerical weather prediction", PhD thesis, Imperial College London, 2020. DOI: 10.25560/86162.

[38] J.-C. Nédélec, "Mixed finite elements in 3", *Numerische Mathematik*, vol. 35, no. 3, pp. 315–341, 1980.

[39] P.-A. Raviart and J.-M. Thomas, "A mixed finite element method for 2-nd order elliptic problems", in *Mathematical aspects of finite element methods*, Springer, 1977, pp. 292–315.

[40] F. Brezzi, J. Douglas, and L. D. Marini, "Two families of mixed finite elements for second order elliptic problems", *Numerische Mathematik*, vol. 47, no. 2, pp. 217–235, 1985.

[41] D. N. Arnold and A. Logg, "Periodic Table of the Finite Elements", *SIAM News*, vol. 47, no. 9, 2014.

[42] Y. Saad and M. H. Schultz, "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems", *SIAM Journal on scientific and statistical computing*, vol. 7, no. 3, pp. 856–869, 1986.

[43] M. R. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*, 1. NBS Washington, DC, 1952, vol. 49.

[44] G. H. Golub and C. F. Van Loan, *Matrix computations*. JHU press, 2013.

[45] J. Betteridge, T. H. Gibson, I. G. Graham, and E. H. Müller, "Multigrid preconditioners for the hybridised discontinuous Galerkin discretisation of the shallow water equations", *Journal of Computational Physics*, vol. 426, p. 109 948, 2021. DOI: 10.1016/j.jcp.2020.109948.

[46] D. N. Arnold and F. Brezzi, "Mixed and nonconforming finite element methods: Implementation, postprocessing and error estimates", *ESAIM: Mathematical Modelling and Numerical Analysis-Modélisation Mathématique et Analyse Numérique*, vol. 19, no. 1, pp. 7–32, 1985.

[47] D. N. Arnold, F. Brezzi, B. Cockburn, and L. D. Marini, "Unified analysis of discontinuous Galerkin methods for elliptic problems", *SIAM journal on numerical analysis*, vol. 39, no. 5, pp. 1749–1779, 2002. DOI: 10.1137/S0036142901384162.

[48] B. Cockburn, J. Gopalakrishnan, and R. Lazarov, "Unified hybridization of discontinuous Galerkin, mixed, and continuous Galerkin methods for second order elliptic problems", *SIAM Journal on Numerical Analysis*, vol. 47, no. 2, pp. 1319–1365, 2009. DOI: /10.1137/070706616.

[49] J. Gopalakrishnan and S. Tan, "A convergent multigrid cycle for the hybridized mixed method", *Numerical Linear Algebra with Applications*, vol. 16, no. 9, pp. 689–714, 2009. DOI: 10.1002/nla.636.

[50] B. Cockburn, O. Dubois, J. Gopalakrishnan, and S. Tan, "Multigrid for an HDG method", *IMA Journal of Numerical Analysis*, vol. 34, no. 4, pp. 1386–1425, 2014. DOI: 10.1093/imanum/drt024.

[51] L. Chen, J. Wang, Y. Wang, and X. Ye, "An auxiliary space multigrid preconditioner for the weak Galerkin method", *Computers & Mathematics with Applications*, vol. 70, no. 4, pp. 330–344, 2015. DOI: 10.1016/j.camwa.2015.04.016.

[52] T. Wildey, S. Muralikrishnan, and T. Bui-Thanh, "Unified geometric multigrid algorithm for hybridized high-order finite element methods", *SIAM Journal on Scientific Computing*, vol. 41, no. 5, S172–S195, 2019. DOI: 10.1137/18M1193505.

[53] S. Muralikrishnan, T. Bui-Thanh, and J. N. Shadid, "A multilevel approach for trace system in HDG discretizations", *Journal of Computational Physics*, vol. 407, p. 109 240, 2020. DOI: 10.1016/j.jcp.2020.109240.

[54] S. A. Orszag, "Spectral methods for problems in complex geometrics", in *Numerical methods for partial differential equations*, Elsevier, 1979, pp. 273–305. DOI: 10.1016/B978-0-12-546050-7.50014-9.

[55] M. Homolya, L. Mitchell, F. Luporini, and D. Ham, "TSFC: A Structure-Preserving Form Compiler", *SIAM Journal on Scientific Computing*, vol. 40, no. 3, pp. C401–C428, 2018. DOI: 10.1137/17M1130642. [Online]. Available: https://doi.org/10.1137/17M1130642.

[56] F. Luporini, A. L. Varbanescu, F. Rathgeber, G.-T. Bercea, J. Ramanujam, D. A. Ham, and P. H. J. Kelly, "Cross-loop optimization of arithmetic intensity for finite element local assembly", *ACM Transactions on Architecture and Code Optimization*, vol. 11, no. 4, 57:1–57:25, 2015. DOI: 10.1145/2687415. [Online]. Available: http://doi.acm.org/10.1145/2687415.

[57] A. Klöckner, "Loo. py: Transformation-based code generation for GPUs and CPUs", in *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ACM, 2014, p. 82. DOI: 10.1145/2627373.2627387.

[58] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, D. Karpeyev, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, "PETSc users manual", Argonne National Laboratory, Tech. Rep. ANL-95/11 - Revision 3.13, 2020. [Online]. Available: https://www.mcs.anl.gov/petsc.

[59] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient management of parallelism in object oriented numerical software libraries", in *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds., Birkhäuser Press, 1997, pp. 163–202. DOI: 10.1007/978-1-4612-1986-6_8.

[60] M. S. Alnæs, "UFL: a finite element form language", in *Automated Solution of Differential Equations by the Finite Element Method*, Springer, 2012, pp. 303–338. DOI: 10.1007/978-3-642-23099-8_17.

[61] T. H. Gibson, L. Mitchell, D. A. Ham, and C. J. Cotter, "Slate: extending Firedrake's domain-specific abstraction to hybridized solvers for geoscience and beyond.", *Geoscientific model development.*, vol. 13, no. 2, pp. 735–761, 2020. DOI: 10.5194/gmd-13-735-2020. arXiv: 1802.00303 [cs.MS]. [Online]. Available: https://arxiv.org/abs/1802.00303.

[62] F. Rathgeber, G. R. Markall, L. Mitchell, N. Loriant, D. A. Ham, C. Bertolli, and P. H. J. Kelly, "PyOP2: A high-level framework for performance-portable simulations on unstructured meshes", in *High Performance Computing, Networking Storage and Analysis, SC Companion:*, Los Alamitos, CA, USA: IEEE Computer Society, 2012, pp. 1116–1123, ISBN: 978-1-4673-3049-7. DOI: 10.1109/SC.Companion.2012.134. [Online]. Available: http://dx.doi.org/10.1109/SC.Companion.2012.134.

[63] G. Guennebaud, B. Jacob, *et al.*, "Eigen", *http://eigen.tuxfamily.org*, 2010.

[64] A. Logg, G. N. Wells, and J. Hake, "DOLFIN: A C++/Python finite element library", in *Automated solution of differential equations by the finite element method*, Springer, 2012, pp. 173–225. DOI: 10.1007/978-3-642-23099-8_10.

[65] M. S. Alnæs, A. Logg, and K.-A. Mardal, "UFC: a finite element code generation interface", in *Automated Solution of Differential Equations by the Finite Element Method*, Springer, 2012, pp. 283–302. DOI: 10.1007/978-3-642-23099-8_16.

[66] S. Verdoolaege, "Isl: An integer set library for the polyhedral model", in *Mathematical Software – ICMS 2010*, K. Fukuda, J. v. d. Hoeven, M. Joswig, and N. Takayama, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 299–302, ISBN: 978-3-642-15582-6. DOI: 10.1007/978-3-642-15582-6_49.

[67] A. Kloeckner. (2014). Loo.py, [Online]. Available: https://documen.tician.de/loopy/ (visited on 03/06/2020).

[68] A. Kloeckner, L. C. Wilcox, and T. Warburton, "Array program transformation with Loo. py by example: high-order finite elements", in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, 2016, pp. 9–16. DOI: 10.1145/2935323.2935325.

[69] A. Kloeckner. (2013). Pymbolic, [Online]. Available: https://documen.tician.de/pymbolic/ (visited on 06/03/2020).

[70] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, Third. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999, ISBN: 0-89871-447-8 (paperback).

[71] G. GNU. (2022). Vector extensions, [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html.

[72] *Software used in 'Composable code generation for high order, compatible finite element methods - Vectorised Slate'*, 2022. DOI: 10.5281/zenodo.7153137. [Online]. Available: https://doi.org/10.5281/zenodo.7153137.

[73] B. Smith, S. Balay, M. Knepley, J. Brown, H. Zhang, L. C. McInnes, S. Zampini, L. Dalcin, K. Rupp, P. Brune, P. Sanan, V. Hapla, J. Sarich, H. Zhang, J. Zhang, D. Karpeyev, R. T. Mills, prj, V. M, dmay23, F. Kong, V. Minden, V. Eijkhout, A. Dener, tisaac, J. E. Roman, S. Kruger, L. Mitchell, M. Lange, and SurtaiHan, *firedrakeproject/petsc: Portable, Extensible Toolkit for Scientific Computation*, version Firedrake_20220506.0, May 2022. DOI: 10.5281/zenodo.6522877. [Online]. Available: https://doi.org/10.5281/zenodo.6522877.

[74] S. Vorderwuelbecke, *Experimental evaluation of Slate optimisations in Firedrake*, Oct. 2022. DOI: 10.5281/zenodo.7153202. [Online]. Available: https://doi.org/10.5281/zenodo.7153202.

[75] L. Mitchell and E. H. Müller, "High level implementation of geometric multigrid solvers for finite element problems: Applications in atmospheric modelling", *Journal of Computational Physics*, vol. 327, pp. 1–18, 2016. DOI: 10.1016/j.jcp.2016.09.037.

[76] K. Goda, "A multistep technique with implicit difference schemes for calculating two-or three-dimensional cavity flows", *Journal of computational physics*, vol. 30, no. 1, pp. 76–95, 1979. DOI: 10.1016/0021-9991(79)90088-3.

[77] S. Vorderwuelbecke, *Performance experiments for fully matrix-free high order hybridised compatible FEM*, version FullyMatfreeHybridisation, Oct. 2022. DOI: 10.5281/zenodo.7157916. [Online]. Available: https://doi.org/10.5281/zenodo.7157916.

[78] PETSc. (2022). KSP linear solver - solving block matrices, [Online]. Available: https://petsc.org/release/docs/manual/ksp/#solving-block-matrices.

[79] J. W. Klop and J. Klop, *Term rewriting systems*. Centrum voor Wiskunde en Informatica, 1990.

[80] *Software used in 'Composable code generation for high order, compatible finite element methods - Matrix-free Slate'*, 2022. DOI: 10.5281/zenodo.7153109. [Online]. Available: https://doi.org/10.5281/zenodo.7153109.

[81] J. Chang, M. S. Fabien, M. G. Knepley, and R. T. Mills, "Comparative study of finite element methods using the time-accuracy-size (TAS) spectrum analysis", *SIAM Journal on Scientific Computing*, vol. 40, no. 6, pp. C779–C802, 2018.

[82] P. J. Roache, "Code verification by the method of manufactured solutions", *J. Fluids Eng.*, vol. 124, no. 1, pp. 4–10, 2002. DOI: 10.1115/1.1436090.

[83] F. Brezzi and M. Fortin, *Mixed and hybrid finite element methods*. Springer Science & Business Media, 2012, vol. 15.

[84] A. Natale, J. Shipton, and C. J. Cotter, "Compatible finite element spaces for geophysical fluid dynamics", *Dynamics and Statistics of the Climate System*, vol. 1, no. 1, Nov. 2016, dzw005, ISSN: 2059-6987. DOI: 10.1093/climsys/dzw005. eprint: https://academic.oup.com/climatesystem/article-pdf/1/1/dzw005/13106953/dzw005.pdf. [Online]. Available: https://doi.org/10.1093/climsys/dzw005.

[85] R. C. Kirby and L. Mitchell, "Solver composition across the PDE/linear algebra barrier", *SIAM Journal on Scientific Computing*, vol. 40, no. 1, pp. C76–C98, 2018. DOI: 10.1137/17M1133208. arXiv: 1706.01346 [cs.MS]. [Online]. Available: http://arxiv.org/abs/1706.01346.

[86] P. D. Brubeck and P. E. Farrell, "A scalable and robust vertex-star relaxation for high-order FEM", *arXiv preprint arXiv:2107.14758*, 2021. DOI: 10.48550/arXiv.2107.14758.

[87] N. A. Loppi, F. D. Witherden, A. Jameson, and P. E. Vincent, "A high-order cross-platform incompressible navier–stokes solver via artificial compressibility with application to a turbulent jet", *Computer Physics Communications*, vol. 233, pp. 193–205, 2018.

[88] A. Jameson, P. E. Vincent, and P. Castonguay, "On the non-linear stability of flux reconstruction schemes", *Journal of Scientific Computing*, vol. 50, no. 2, pp. 434–445, 2012.

[89] P. E. Vincent, P. Castonguay, and A. Jameson, "A new class of high-order energy stable flux reconstruction schemes", *Journal of Scientific Computing*, vol. 47, no. 1, pp. 50–72, 2011.

[90] H. Salehipour and W. Peltier, "Diapycnal diffusivity, turbulent prandtl number and mixing efficiency in boussinesq stratified turbulence", *Journal of Fluid Mechanics*, vol. 775, 464–500, 2015. DOI: 10.1017/jfm.2015.305.

[91] A Mashayek, H Salehipour, D Bouffard, C. Caulfield, R Ferrari, M Nikurashin, W. Peltier, and W. Smyth, "Efficiency of turbulent mixing in the abyssal ocean circulation", *Geophysical Research Letters*, vol. 44, no. 12, pp. 6296–6306, 2017. DOI: 10.1002/2016GL072452.

Appendix

A Contributions: Pull requests (PR)

The contributions of the thesis listed in 1.3 are linked to the corresponding pull requests in the main codebase.

1. *TSSLAC*: Firedrake PR #1651.
2. *Slate Vectorisation*: PyOP2 PR #654
3. *Matrix-free methods in Slate*
   (a) Change of the assembly strategy of blocks: Firedrake PR #2111
   Reordering of the Slate expression: Firedrake PR #2233.
   (b) Local preconditioners: Firedrake PR #2259.
   (c) Local actions and local matrix-free solvers: Firedrake PR #2288.
4. *Nesting of Schur complements in the Hybridisation preconditioner*: Firedrake PR #2237.
5. *Local profiling infrastructure*: PyOP2 PR #658.

B Bandwidth benchmark

In order to get a realistic peak bandwidth limit for the machine used in section 3.4, I ran the STREAM triad benchmark. The code was provided by Jack Betteridge and can be found in `https://github.com/sv2518/system_profiling/tree/my-results/stream`. The benchmark is run for a variety of cores to make sure to get a realisitic limit. For example, running with 2 cores only would not give a realistic peak, as becomes clear from figure 9.1.

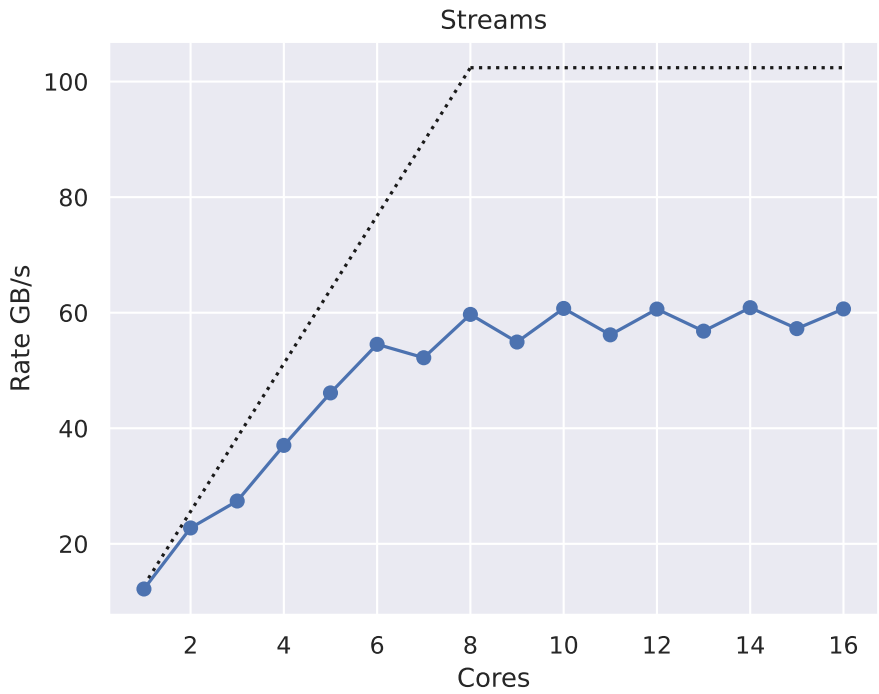


Figure 9.1: STREAM triad benchmark on the hardware specified in section 3.4 run on a variety of cores.

C Slate vectorisation

C.1 Setup

All experiments are run in parallel with hyperthreading by driving the runs with the following command. `mpiexec -np 32 -bind-to "hwthread" -map-by "hwthread"`. The script is run with 32 cores since there are 2 sockets with 8 cores each of which has 2 hardware threads.

The amount of elements in the mesh depend on the polynomial degree of the approximation and the type of mesh.

p	**tri**	**quad**	**tet**	**hex**
$p <= 2$	2048	2048	64	64
$2 < p <= 4$	1024	2048	64	64
$4 < p <= 7$	512	1024	32	64
$7 < p <= 9$	512	1024	32	32
$9 < p$	256	512	32	32

Table 9.1: The size N of the meshes used for the vectorisation results in chapter 3.4. 2D-meshes are generated as $N \times N$ mesh and 3d-meshes as $N \times N \times N$ mesh, both of length N

C.2 Results in table for vectorised Slate expressions

		tri		quad		tet		hex	
	P	AI	S	AI	S	AI	S	AI	S
mass	1	1.8	1.7	5.1	1.7	4.0	1.5	17.6	1.2
	2	2.0	1.7	4.2	1.6	6.4	1.6	11.1	1.1
	3	3.2	1.6	4.1	1.5	9.0	1.2	8.7	1.5
	4	5.8	1.3	4.1	1.9	39.4	1.0	7.6	1.4
	5	7.7	1.2	4.1	1.2	56.0	0.8	7.2	1.5
	6	9.9	1.0	4.3	1.7	81.4	1.2	7.0	1.5
helmholtz	1	8.7	1.7	33.1	1.5	22.0	1.5	138.1	1.5
	2	17.8	1.9	32.4	1.1	83.8	2.1	105.0	1.3
	3	29.2	2.0	31.8	1.4	113.6	1.5	83.8	1.3
	4	53.7	2.3	31.6	2.2	492.8	1.7	73.5	2.0
	5	70.3	1.5	31.5	2.0	690.2	2.0	69.4	1.9
	6	90.0	1.3	32.8	2.2	993.9	3.6	68.0	2.0
laplacian	1	5.6	1.7	24.8	1.7	19.6	1.4	114.6	1.4
	2	8.6	1.7	26.6	1.1	32.5	1.8	81.4	1.5
	3	12.5	1.9	25.4	1.3	72.8	2.0	64.7	1.2
	4	20.9	2.4	25.1	2.1	94.9	2.4	57.6	1.7
	5	38.2	1.9	24.9	2.1	372.9	2.3	55.1	1.7
	6	50.7	1.9	25.9	2.6	567.9	2.2	54.8	1.7
elasticity	1	6.7	1.7	31.8	1.7	24.0	1.5	145.2	1.4
	2	9.7	1.7	31.0	1.1	36.3	1.9	95.4	1.5
	3	13.6	1.9	28.6	1.2	77.6	2.1	73.3	1.2
	4	22.1	2.4	27.7	2.0	98.7	2.4	64.0	1.7
	5	39.9	1.9	27.1	2.1	382.4	2.4	60.4	1.6
	6	52.4	1.9	28.0	2.3	577.7	2.2	59.3	1.7
hyperelasticity	1	11.5	1.7	96.9	1.4	41.2	1.5	548.1	0.9
	2	29.9	2.0	94.3	1.7	190.8	2.7	412.1	1.7
	3	80.8	3.0	92.1	1.2	942.0	2.6	354.9	1.1
	4	122.4	2.6	92.0	2.6	1800.8	2.5	330.1	2.1
	5	168.7	2.3	91.6	2.5	2743.8	2.2	323.6	2.2
	6	223.9	2.0	95.4	2.8	4282.2	2.1	325.7	2.3

Table 9.2: Arithmetic intensity (AI) and speedup (S) of explicit vectorisation over auto-vectorisation for a Slate expression for a variety of operators discretised with CG on different meshes. The setup is explained in 3.4.2.

C.3 Results in table for vectorised actions

		tri				quad				tet				hex			
	P	AI	D	Q	S	AI	D	Q	S	AI	D	Q	S	AI	D	Q	S
mass	1	1.5	3	3	2.1	4.8	2	3	0.5	3.5	4	4	1.2	17.3	2	3	1.1
	2	1.8	6	6	1.0	4.0	3	4	0.6	6.1	10	14	1.3	10.8	3	4	1.0
	3	3.0	10	12	1.1	3.9	4	5	0.5	8.8	20	24	1.0	8.5	4	5	1.5
	4	5.7	15	25	1.7	3.9	5	6	1.4	39.2	35	125	1.0	7.4	5	6	1.5
	5	7.5	21	36	8.4	3.9	6	7	1.6	55.8	56	216	0.8	7.0	6	7	1.5
	6	9.7	28	49	1.4	4.1	7	8	1.1	81.2	84	343	1.1	6.9	7	8	1.6
helmholtz	1	2.8	3	3	1.6	10.9	2	3	1.3	7.2	4	4	1.2	45.9	2	3	1.2
	2	5.9	6	6	1.7	10.7	3	4	1.0	27.8	10	14	1.8	34.9	3	4	1.1
	3	9.7	10	12	1.7	10.5	4	5	1.0	37.8	20	24	1.3	27.9	4	5	1.2
	4	17.9	15	25	2.1	10.5	5	6	1.8	164.2	35	125	1.7	24.5	5	6	1.6
	5	23.4	21	36	1.3	10.4	6	7	2.1	230.0	56	216	1.7	23.1	6	7	1.8
	6	30.0	28	49	1.2	10.9	7	8	1.9	331.3	84	343	1.6	22.6	7	8	2.0
laplacian	1	1.7	3	1	1.4	8.2	2	3	1.5	6.2	4	1	1.2	38.0	2	3	1.2
	2	2.8	6	3	1.5	8.8	3	4	0.9	10.7	10	4	1.4	27.0	3	4	1.3
	3	4.1	10	6	1.5	8.4	4	5	1.0	24.1	20	14	1.7	21.5	4	5	1.2
	4	6.9	15	12	1.8	8.3	5	6	2.1	31.5	35	24	2.2	19.1	5	6	1.5
	5	12.7	21	25	1.5	8.2	6	7	2.1	124.2	56	125	2.3	18.3	6	7	1.6
	6	16.9	28	36	1.5	8.6	7	8	2.2	189.2	84	216	2.1	18.2	7	8	1.6
elasticity	1	2.1	3	1	2.4	10.5	2	3	1.5	7.7	4	1	1.1	48.2	2	3	1.3
	2	3.1	6	3	1.5	10.3	3	4	1.2	11.9	10	4	1.5	31.7	3	4	1.2
	3	4.5	10	6	1.0	9.5	4	5	1.1	25.7	20	14	1.7	24.4	4	5	1.2
	4	7.3	15	12	1.9	9.2	5	6	1.9	32.8	35	24	2.2	21.3	5	6	1.7
	5	13.2	21	25	1.5	9.0	6	7	1.9	127.4	56	125	2.3	20.1	6	7	1.6
	6	17.4	28	36	1.6	9.3	7	8	2.0	192.5	84	216	2.1	19.7	7	8	1.7
hyperelasticity	1	3.7	3	1	1.3	32.2	2	4	1.1	13.5	4	1	1.2	182.6	2	4	0.9
	2	9.9	6	6	2.0	31.4	3	6	1.8	63.5	10	14	2.1	137.3	3	6	1.7
	3	26.9	10	25	2.7	30.6	4	8	1.2	313.9	20	125	2.5	118.2	4	8	1.1
	4	40.8	15	49	2.6	30.6	5	10	2.4	600.2	35	343	2.5	110.0	5	10	2.3
	5	56.2	21	81	2.2	30.5	6	12	2.2	914.6	56	729	2.1	107.8	6	12	2.3
	6	74.6	28	121	1.9	31.8	7	14	2.7	1427.3	84	133	2.0	108.5	7	14	2.3

Table 9.3: Arithmetic intensity (AI) and speedup (S) of explicit vectorisation over auto-vectorisation for an action on variety of operators discretised with CG on different meshes through TSFC. The setup is explained in 3.4.2

D Numerical investigation: Outer iterations

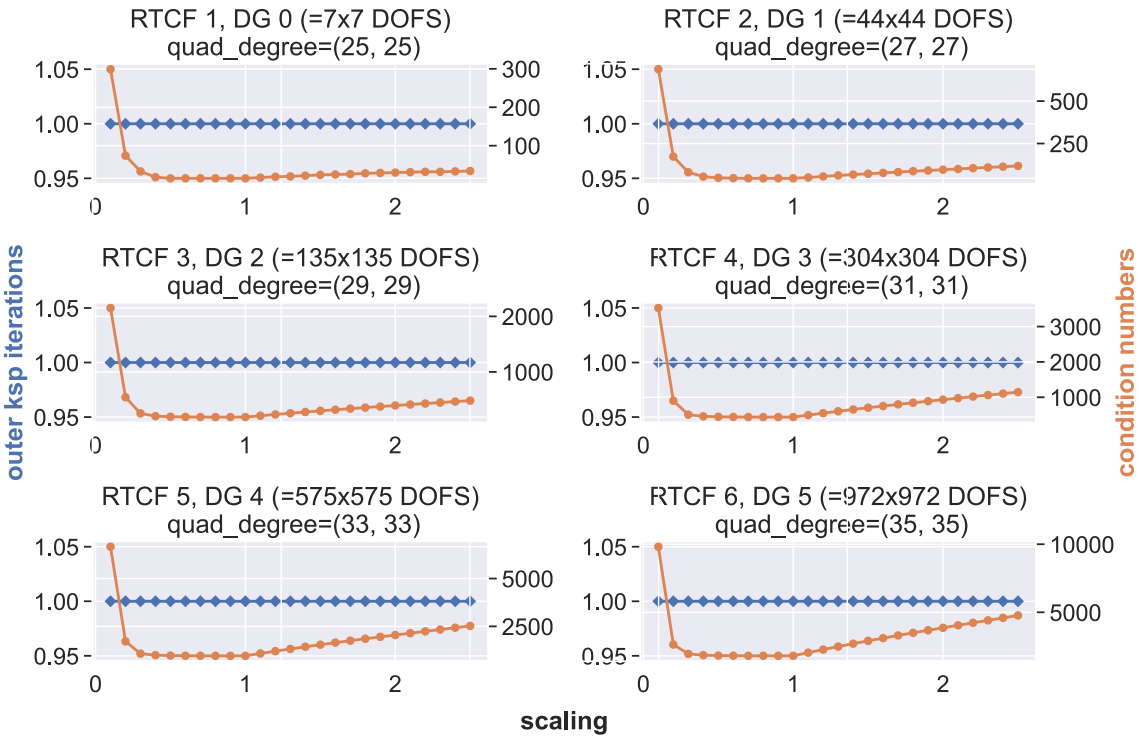


Figure 9.2: Cell scalings: Outer iterations

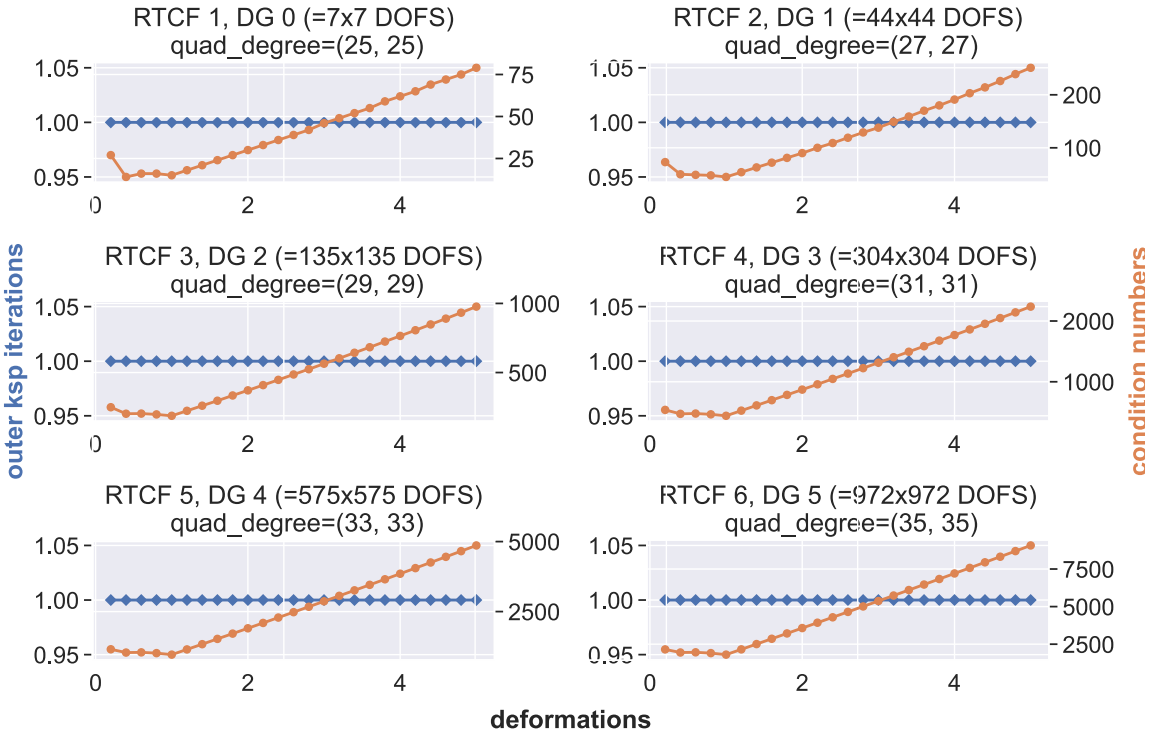


Figure 9.3: Affine cell deformations: Outer iterations

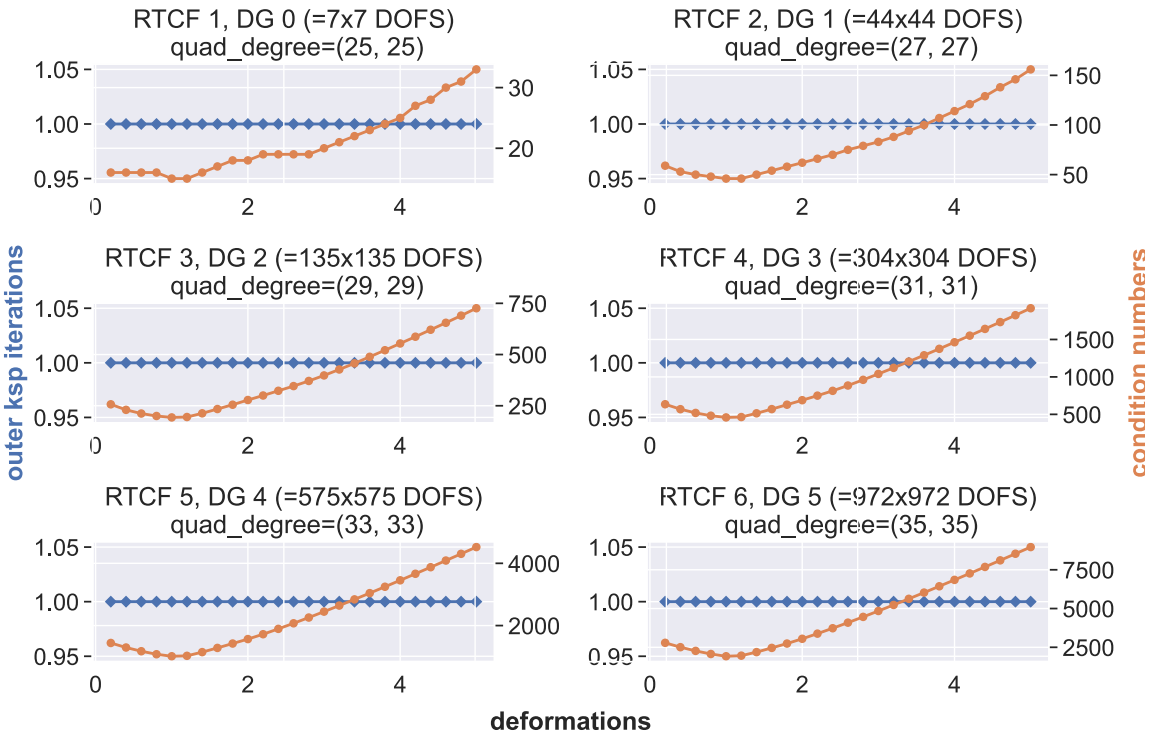


Figure 9.4: Non-affine cell deformations: Outer iterations

E Matrix-free hybridisation: Solvers Diagrams

E.1 GTMG

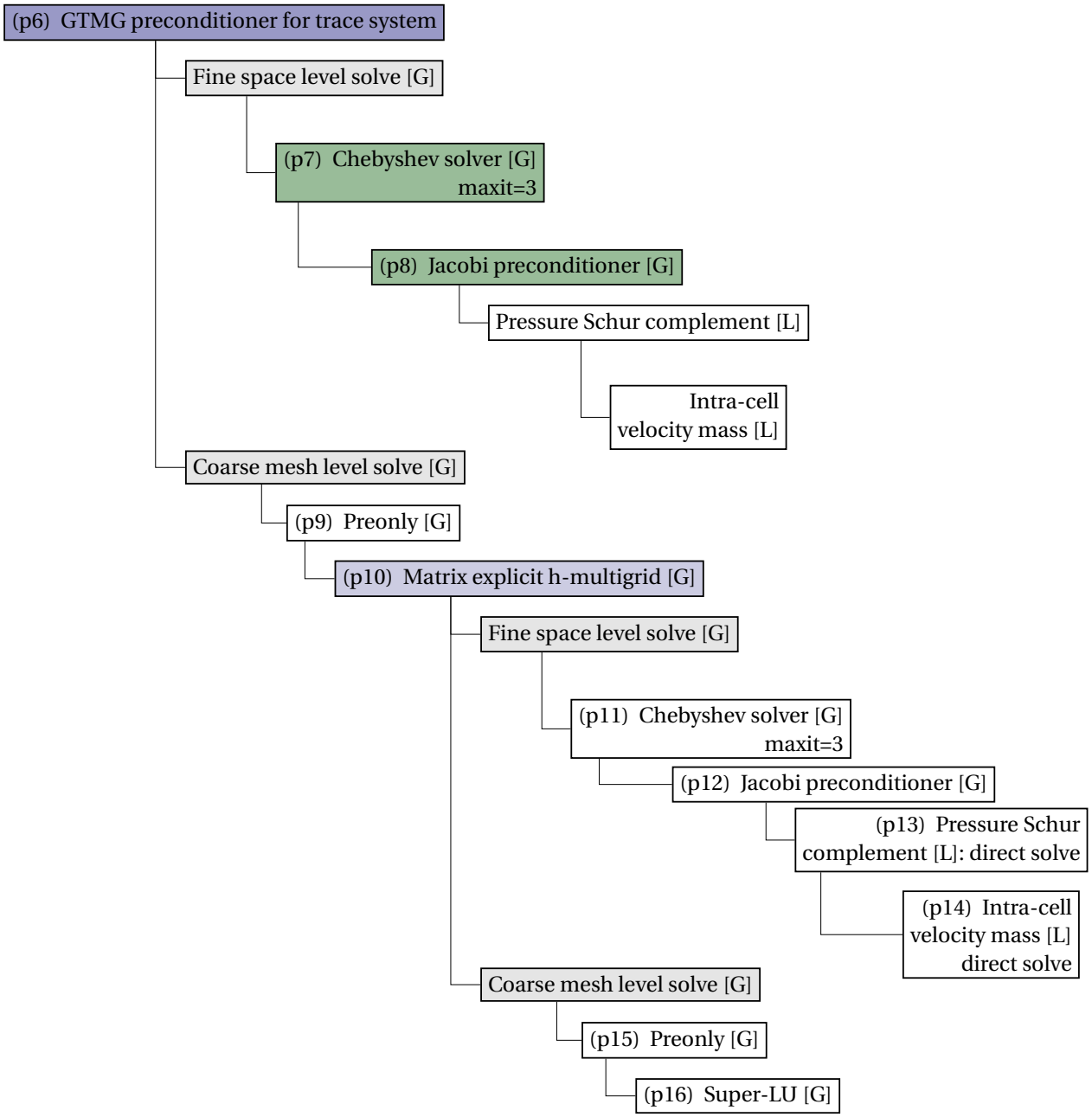


Figure 9.5: Setup of the GTMG preconditioner. The diagram is zooming into the solver option specified in node (p35), (p41) and (p47) in following solvers diagram 5.1. The matrix types in green nodes changes depending on the case. For case 2 node (p11) and (p12) are matrix-explicit. For case 4, 5 and 6 node (p11) is matrix-free and (p12) is matrix-explicit. The local solvers on the fine space level solve are specified in the following, full solver diagrams and depend on the case.

E.2 Solver for case 1

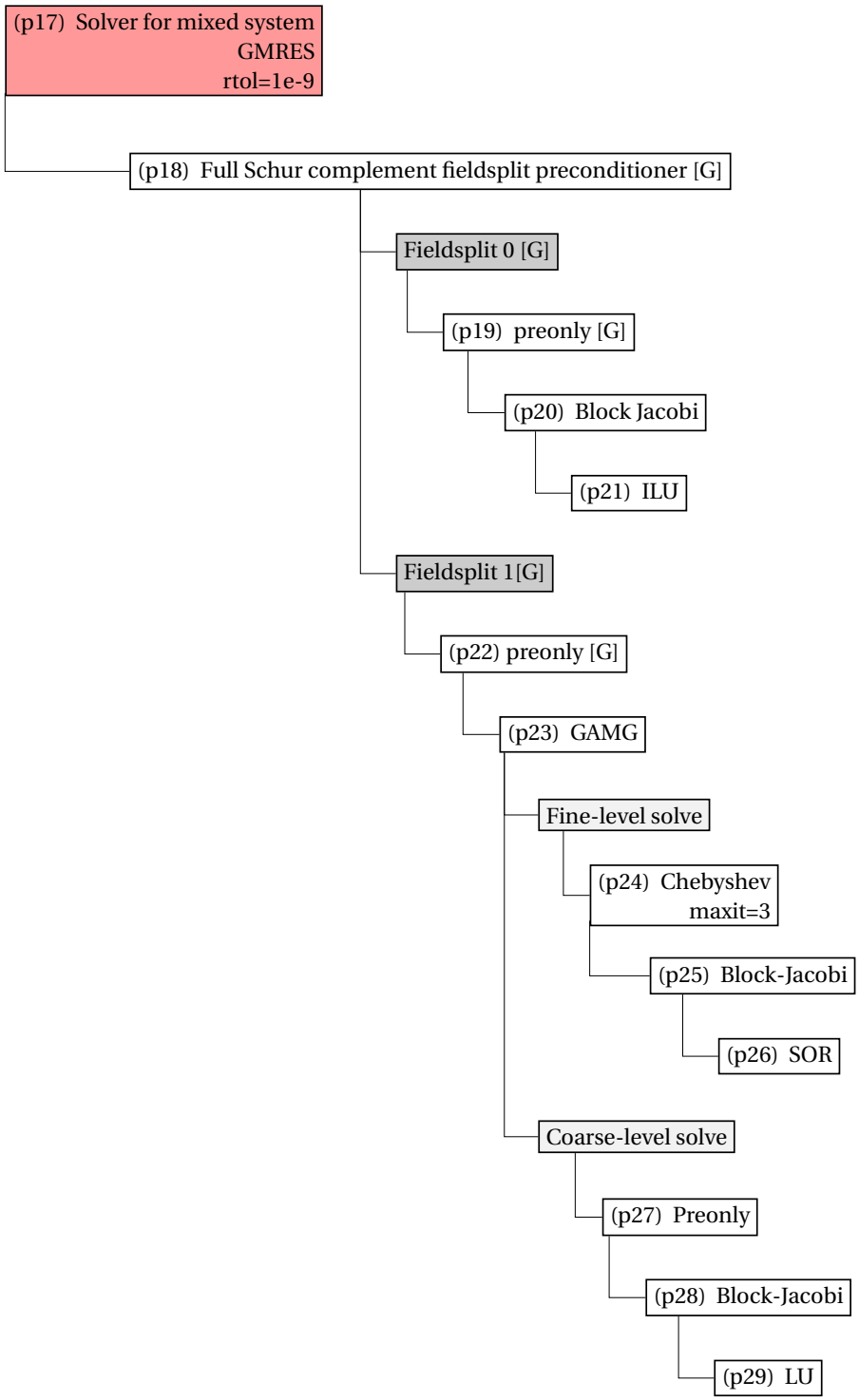


Figure 9.6: Solver setup for the case 1. Red denotes the outermost solve, gray separates parts of the fieldsplit preconditioner. Note that all solves are global.

E.3 Solver for case 2 to 5

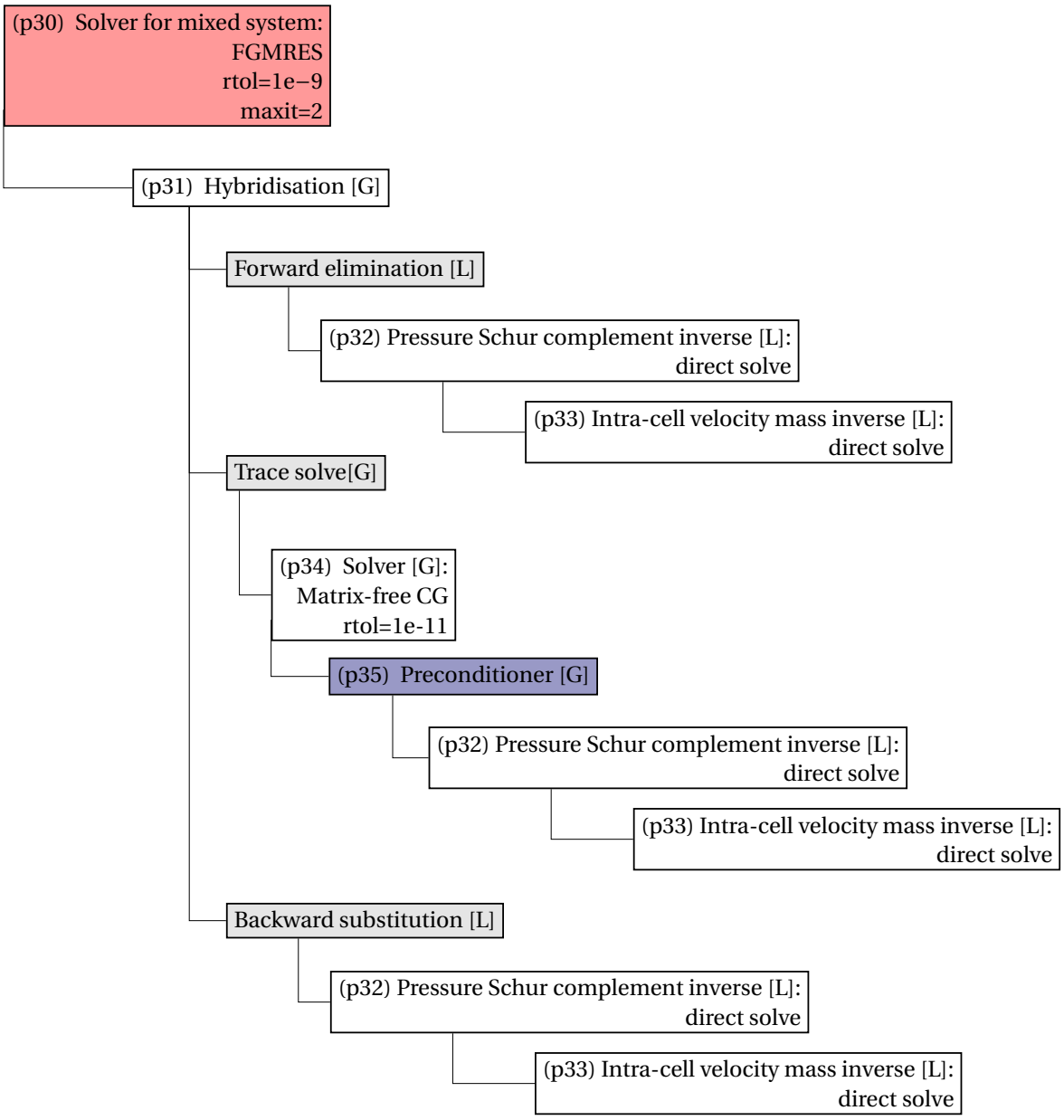


Figure 9.7: General diagram of the solver setup for the case 2-5. The forward elimination and backward substitution are purely local kernels (denoted by [L]), whereas the trace solve involves global kernels (denoted by [G]) and local ones. Red denotes the outermost solve, gray separates parts of the hybridisation preconditioner, blue denotes the trace preconditioner. For case2 the node (p35) is a Jacobi preconditioner,for case 4 it is assembled Jacobi, and for case 3 it is matrix-explicit GTMG and for case 5 it is globally matrix-free GTMG 9.5.

E.4 Solver for case 6

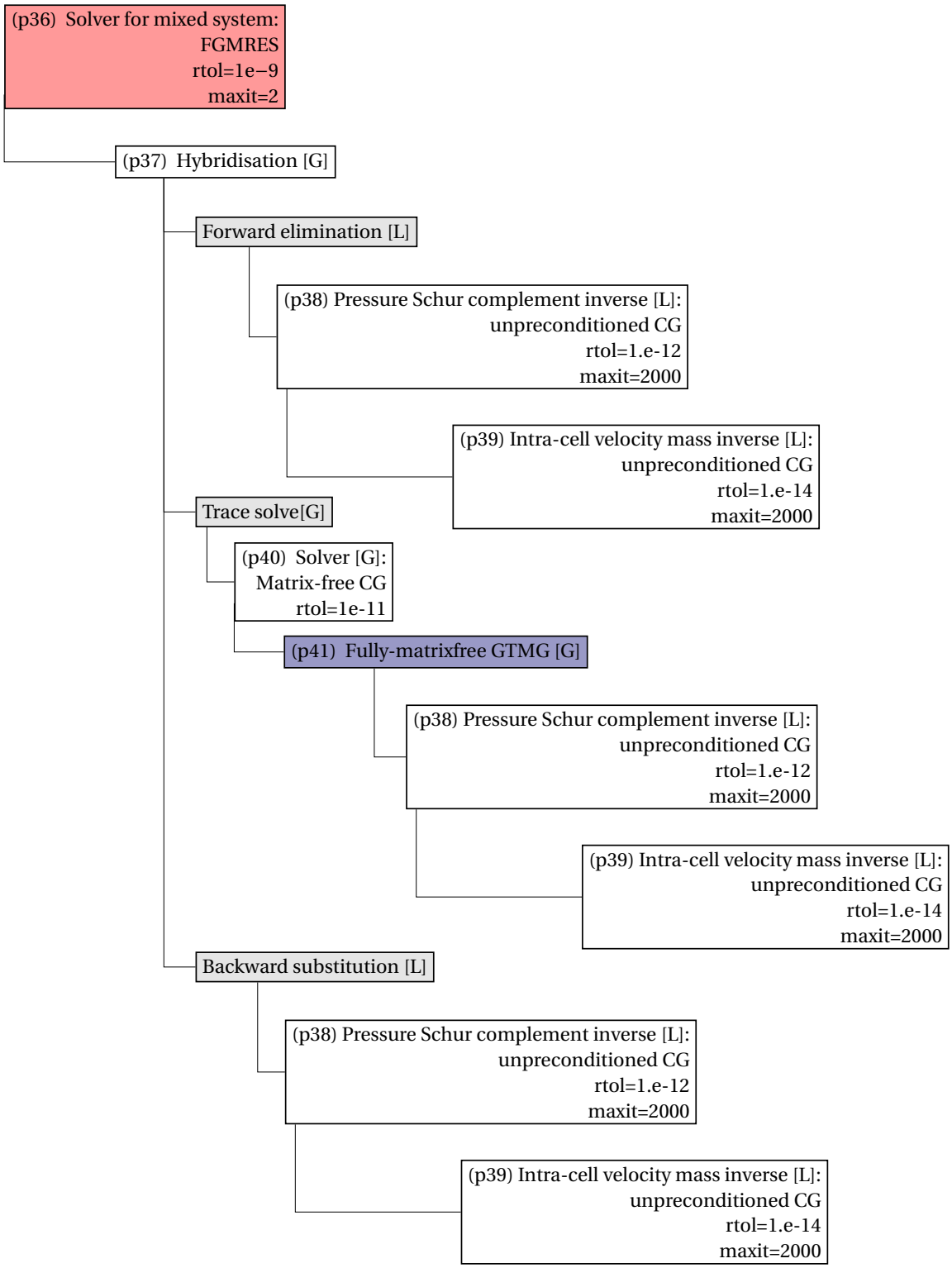


Figure 9.8: General diagram of the solver setup for the case 6. Red denotes the outermost solve, gray separates parts of the hybridisation preconditioner, blue denotes the trace preconditioner in its fully matrix-free form 9.5.

E.5 Solver for case 7

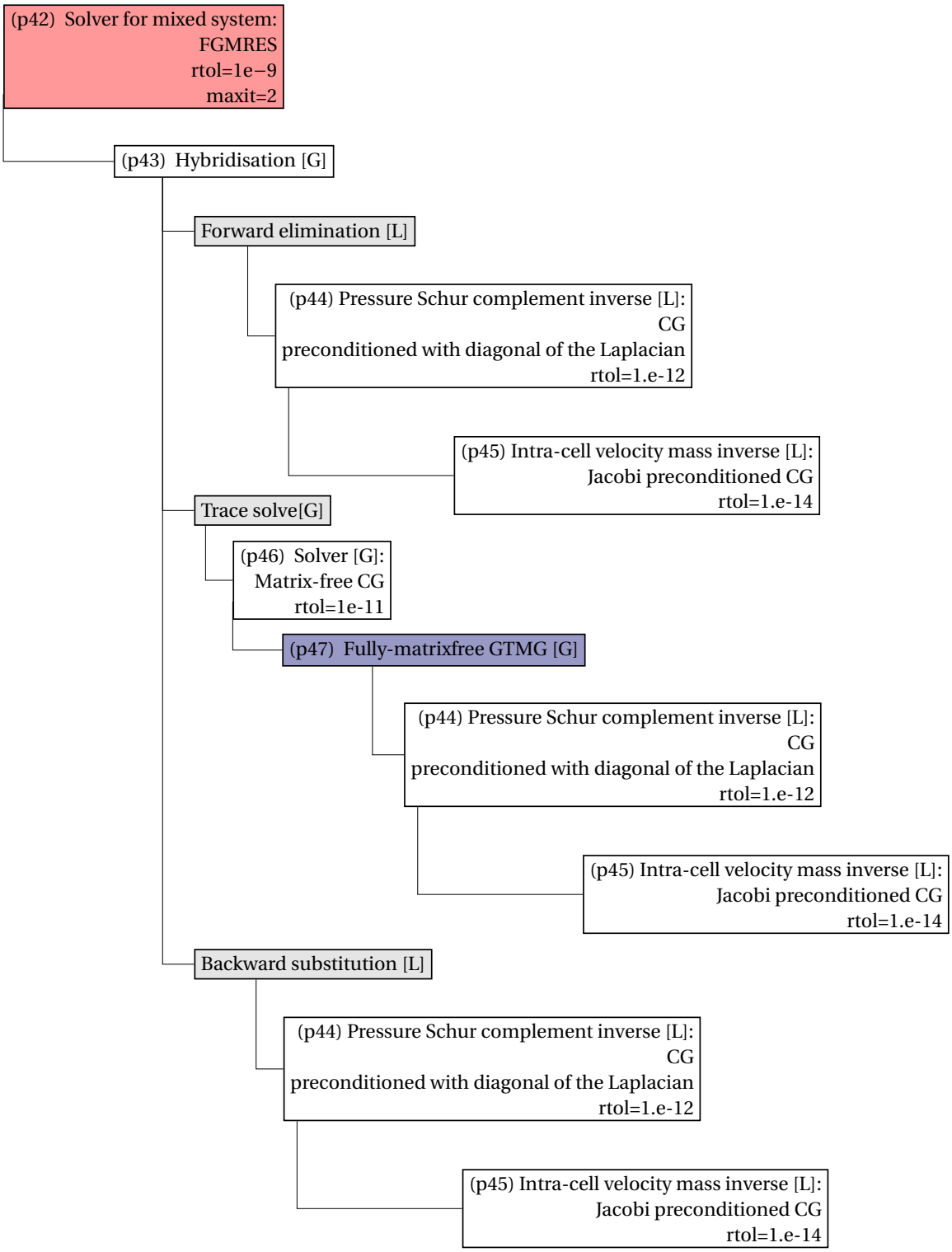


Figure 9.9: General diagram of the solver setup for the case 7. Red denotes the outermost solve, gray separates parts of the hybridisation preconditioner, blue denotes the trace preconditioner in its fully matrix-free form 9.5.

F Matrix-free hybridisation: Heatmaps

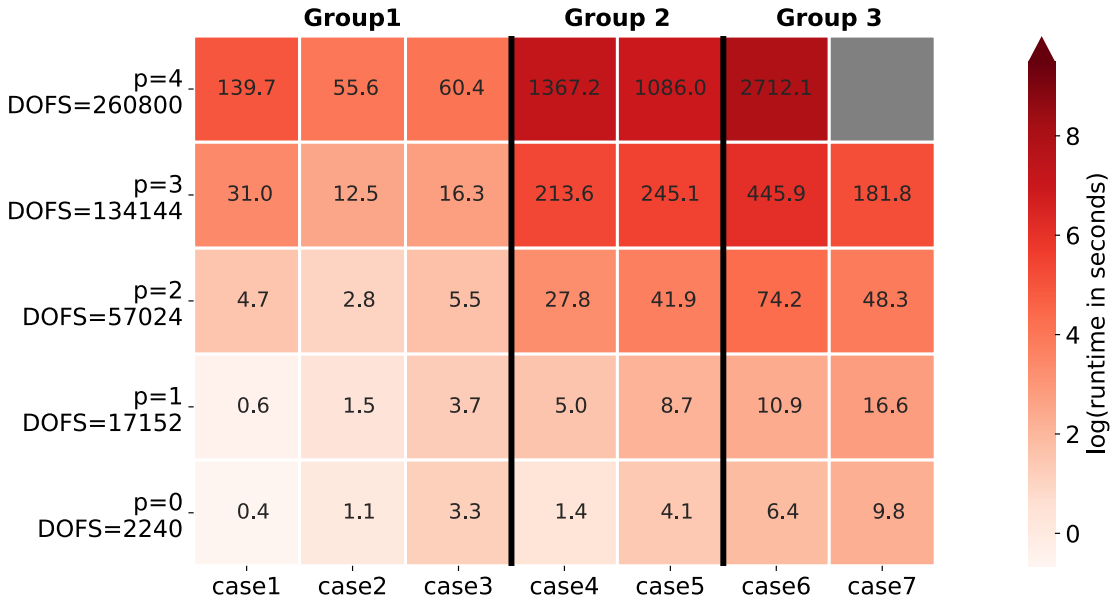


Figure 9.10: Heatmap of runtime performance measurements in seconds for solving the mixed Poisson problem on a fixed mesh ($8 \times 8 \times 8$). The cases for different solver setups can be found in 5.1. Gray, unannotated cells correspond to a solver which is not able to solve the problem.

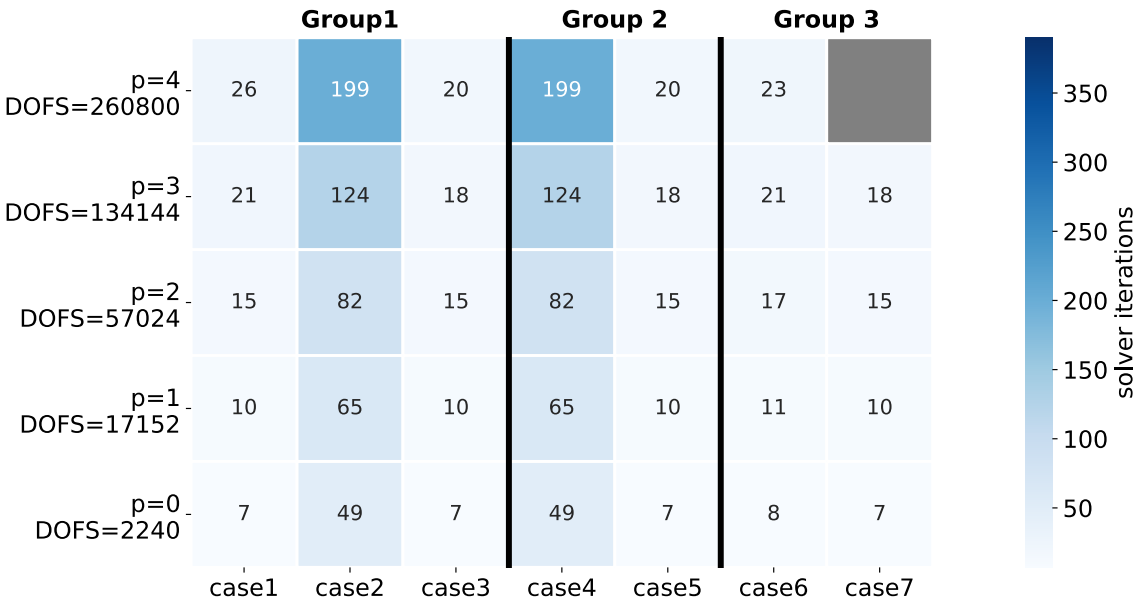


Figure 9.11: Heatmap of outer solver iterations (for case 1) and trace solver iterations (for the other cases) for solving the mixed Poisson problem on a fixed mesh ($8 \times 8 \times 8$). In case 1, the reference solver, is not using hybridisation and has therefore no trace iterations, but outer iterations. The cases for different solver setups can be found in 5.1. Gray, unannotated cells correspond to a solver which is not able to solve the problem.