Imperial College London
Department of Computing

Bho

Fabio Luporini

September 2014

Supervised by: Dr. David A. Ham, Prof. Paul H. J. Kelly

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

Declaration

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

Fabio Luporini

Abstract

TODO

Contents

1 Introduction 1

1.1 Overview 1

1.2 Contributions 2

1.3 Thesis Statement 3

1.4 Motivation 3

1.5 Contributions 3

1.6 Statement of Originality 3

1.7 Dissemination 3

1.8 Thesis Outline 3

2 Background 5

2.1 bho 5

3 Sparse Tiling 7

3.1 bho 7

4 Cross-loop Optimization of Arithmetic Intensity for Finite Element Local Assembly 9

4.1 Introduction and Motivations 9

4.2 Preliminaries 14

4.2.1 Overview of the Finite Element Method 15

4.2.2 Quadrature Representation for Finite Element Local Assembly 16

4.2.3 Implementation of Quadrature-based Local Assembly 17

4.3 Overview of the Optimization Strategy 22

4.4 Expression Rewriting 23
  4.4.1 Generalized Loop-invariant Code Motion 24
  4.4.2 Terms Factorization 26
  4.4.3 Precomputation of Invariant Terms 27
  4.4.4 Expanding Sub-expressions 28
  4.4.5 Rewrite Rules for Assembly Expressions 30
  4.4.6 Avoiding Iteration over Zero-valued Blocks by Symbolic Execution 32
4.5 Code Specialization 35
  4.5.1 Padding and Data Alignment 35
  4.5.2 Expression Splitting 36
  4.5.3 Model-driven Vector-register Tiling 39
  4.5.4 Exposing Matrix-Matrix Multiplications for BLAS Operations 40
4.6 General-purpose Optimizations 42
4.7 Design and Implementation of COFFEE 42
  4.7.1 Input ad Output: the Integration with Firedrake . . . 42
  4.7.2 Structure 42
  4.7.3 Conveying Domain-Specific Knowledge 42
  4.7.4 Model-driven Dynamic Autotuning 42
4.8 Performance Analysis 45
  4.8.1 Contribution of Individual Optimizations 45
  4.8.2 Evaluation in Forms of Increasing Complexity 45
  4.8.3 Details on the Autotuning Process 45
  4.8.4 Full Application Study 45
4.9 Related Word 45
4.10 Generality of the Approach and Applicability to Other Domains 45
4.11 Conclusion 45

5 Conclusion **47**
5.1 bho 47

List of Tables

4.1 Type and variable names used in the various listings to identify local assembly objects. 20

List of Figures

4.1 Structure of a local assembly kernel 19
4.2 Original (simplified) code 24
4.3 Invariant code 25
4.4 Factorized code 26
4.5 Scalar-expanded code 27
4.6 Expansion of terms to improve register pressure in a local assembly kernel 29
4.7 Rewrite rules driving the rewriting process of an assembly expression. 31
4.8 Simplified excerpt of local assembly code from a Burgers form using vector-valued basis functions, before and after symbolic execution is performed to rewrite the iteration space 33
4.9 Local assembly code for the Burgers problem in Figure 4.6(c) after the application of *split*. In this example, the split factor is 2. 37
4.10 Local assembly code for the Burgers problem in Figure 4.8(a) after the application of vector-register tiling (outer-product vectorization). In this example, the unroll-and-jam factor is 1. 38
4.11 Outer-product vectorization by permuting values in a vector register. 40
4.12 Restoring the storage layout after *op-vect*. The figure shows how 4×4 elements in the top-left block of the element matrix A can be moved to their correct positions. Each rotation, represented by a group of three same-colored arrows, is implemented by a single shuffle intrinsic. 40

Chapter 1

Introduction

1.1 Overview

In many fields, such as computational fluid dynamics, computational electromagnetics and structural mechanics, phenomena are modelled by partial differential equations (PDEs). Unstructured meshes, which allow an accurate representation of complex geometries, are often used to discretize their computational domain. Numerical techniques, like the finite volume method and the finite element method, approximate the solution of a PDE by applying suitable numerical operations, or kernels, to the various entities of the unstructured mesh, such as edges, vertices, or cells. On standard clusters of multicores, typically, a kernel is executed sequentially by a thread, while parallelism is achieved by partitioning the mesh and assigning each partition to a different node or thread. Such an execution model, with minor variations, is adopted, for instance, in Markall et al. [2013], Logg et al. [2012], AMCG [2010], DeVito et al. [2011], which are examples of frameworks specifically thought for writing numerical methods for PDEs.

The time required to execute these unstructured-mesh-based applications is a fundamental issue. An equation domain needs to be discretized into an extremely large number of cells to obtain a satisfactory approximation of the solution, possibly of the order of trillions (e.g. Rossinelli et al. [2013]), so applying numerical kernels all over the mesh is expensive. For example, it is well-established that mesh resolution is crucial in the accuracy of numerical weather forecasts; however, operational centers have a strict time limit in which to produce a forecast - 60 minutes in the case of the UK

Met Office - so, executing computation- and memory-efficient kernels has a direct scientific payoff in higher resolution, and therefore more accurate predictions. Motivated by this and analogous scenarios, this thesis studies, formalizes, and implements a number of code transformations to improve the performance of real-world scientific applications using numerical methods over unstructured meshes.

1.2 Contributions

Besides developing novel compilers theory, contributions of this thesis come in the form of tools that have been, and currently are, used to accelerate scientific computations, namely:

- **A run-time library, which can be regarded as a prototype compiler, capable of optimizing sequences of non-affine loops by fusion and automatic parallelization. The library has been used to speed up a real application for tsunami simulations as well as other representative benchmarks.**

  One limiting factor to the performance of unstructured mesh applications is imposed by the need for indirect memory accesses (e.g. `A[B[i]]`) to read and write data associated with the various entities of the discretized domain. For example, when executing a kernel that numerically computes an integral over a cell, which is common in a finite element method, it may be necessary to read the coordinates of the adjacent vertices; this is achieved by using a suitable indirection array, often referred to as "map", that connects cells to vertices. The problem with indirect memory accesses is that they break several hardware and compiler optimizations, including prefetching and standard loop blocking (or tiling). A first contribution of this thesis is the formalization and development of a technique, called "generalized sparse tiling", that aims at increasing data locality by fusing loops in which indirections are used. The challenges are 1) to determine if and how a sequence of loops can be fused, and 2) doing it efficiently, since the fusability analysis must be performed at run-time, once the maps are available (i.e. once the mesh topology has been read from disk).

- **A fully-operating compiler, named COFFEE[1], which optimizes numerical kernels solving partial differential equations using the finite element method. The compiler is integrated with the Firedrake system (Firedrake contributors [2014]).**

  The second contribution, in the context of the finite element method, is about optimizing the computation of so called local element matrices, which can be responsible for a significant fraction of the overall computation run-time. This is a well-known problem, and several studies can be found in the literature, among which Russell and Kelly [2013], Olgaard and Wells [2010], Knepley and Terrel [2013], Kirby et al. [2005]. With respect to these studies, we propose a novel set of composable, model-aware code transformations explicitly targeting, for the first time, instruction-level parallelism - with emphasis on SIMD vectorization - and register locality. These transformations are automated in COFFEE.

1.3 Thesis Statement

1.4 Motivation

1.5 Contributions

1.6 Statement of Originality

1.7 Dissemination

1.8 Thesis Outline

---

[1] COFFEE stands for COmpiler For FinitE Element local assembly.

Chapter 2

Background

2.1 bho

Chapter 3

Sparse Tiling

3.1 bho

Chapter 4

Cross-loop Optimization of Arithmetic Intensity for Finite Element Local Assembly

4.1 Introduction and Motivations

In many fields such as computational fluid dynamics, computational electromagnetics and structural mechanics, phenomena are modelled by partial differential equations (PDEs). Numerical techniques, like the finite volume method and the finite element method, are widely employed to approximate solutions of these PDEs. Unstructured meshes are often used to discretize the computational domain, since they allow an accurate representation of complex geometries. The solution is sought by applying suitable numerical operations, or kernels, to the entities of a mesh, such as edges, vertices, or cells. On standard clusters of multicores, typically, a kernel is executed sequentially by a thread, while parallelism is achieved by partitioning the mesh and assigning each partition to a different node or thread. Such an execution model, with minor variations, is adopted, for example, in Markall et al. [2013], Logg et al. [2012], AMCG [2010], DeVito et al. [2011].

The time required to apply the numerical kernels is a major issue, since the equation domain needs to be discretized into an extremely large number of cells to obtain a satisfactory approximation of the PDE, possibly of the order of trillions, as in Rossinelli et al. [2013]. For example, it has been well established that mesh resolution is critical in the accuracy of numerical weather forecasts. However, operational forecast centers have a strict time limit in which to produce a forecast - 60 minutes in the case of the UK Met Office. Producing efficient kernels has a direct scientific payoff in higher resolution, and therefore more accurate, forecasts. Computational cost is a dominant problem in computational science simulations, especially for those based on finite elements, which are the subject of this work. In this chapter, we address, in particular, the well-known problem of optimizing the local assembly phase of the finite element method, which can be responsible for a significant fraction of the overall computation run-time, often in the range 30-60% [Russell and Kelly, 2013], [Olgaard and Wells, 2010], [Knepley and Terrel, 2013], [Kirby et al., 2005].

During the assembly phase, the solution of the PDE is approximated by executing a problem-specific kernel over all cells, or elements, in the discretized domain. We restrict our focus to relatively low order finite element methods, in which an assembly kernel's working set is usually small enough to fit the L1 cache. Low order methods are by no means exotic: they are employed in a wide variety of fields, including climate and ocean modeling, computational fluid dynamics, and structural mechanics. The efficient assembly of high order methods such as the spectral element method [Vos et al., 2010] requires a significantly different loop nest structure. High order methods are therefore excluded from our study.

An assembly kernel is characterized by the presence of an affine, often non-perfect loop nest, in which individual loops are rather small: their trip count rarely exceeds 30, and may be as low as 3 for low order methods. In the innermost loop, a problem-specific, compute intensive expression evaluates a two dimensional array, representing the result of local assembly in an element of the discretized domain. With such a kernel structure, we focus on aspects like the minimization of floating-point operations, register allocation and instruction-level parallelism, especially in the form of SIMD vectorization.

We aim to maximize our impact on the platforms that are realistically used for finite element applications, so we target conventional CPU architectures rather than GPUs. The key limiting factor to the execution on GPUs is the stringent memory requirements. Only relatively small problems fit in a GPU memory, and support for distributed GPU execution in general purpose finite element frameworks is minimal. There has been some research on adapting local assembly to GPUs (mentioned later), although it differs from ours in several ways, including: (i) not relying on automated code generation from a domain-specific language (explained next), (ii) testing only very low order methods, (iii) not optimizing for cross-loop arithmetic intensity (the goal is rather effective multi-thread parallelization). In addition, our code transformations would drastically impact the GPU parallelization strategy, for example by increasing a thread's working set. For all these reasons, a study on extending the research to GPU architectures is beyond the scope of this work. In Section 4.10, however, we provide some intuitions about this research direction.

Achieving high-performance on CPUs is non-trivial. The complexity of the mathematical expressions, often characterized by a large number of operations on constants and small matrices, makes it hard to determine a single or specific sequence of transformations that is successfully applicable to all problems. Loop trip counts are typically small and can vary significantly, which further exacerbates the issue. We will show that traditional vendor compilers, such as *GNU's* and *Intel's*, fail at exploiting the structure inherent such assembly expressions. Polyhedral-model-based source-to-source compilers, for instance Bondhugula et al. [2008], can apply aggressive loop optimizations, such as tiling, but these are not particularly helpful in our context, as explained next.

We focus on optimizing the performance of local assembly operations produced by automated code generation. This technique has been proved successful in the context of the FEniCS [Logg et al., 2012] and Firedrake [Firedrake contributors, 2014] projects, become incredibly popular over the last years. In these frameworks, a mathematical model is expressed at high-level by means of a domain-specific language and a domain-specific compiler is used to produce a representation of local assembly operations (e.g. C code). *Our aim is to obtain close-to-peak performance in all of the local assembly operations that such frameworks can produce.* Since the domain-specific language exposed to the users provide as constructs generic differential operators, an incredibly vast set of PDEs, possibly arising in completely different domains, can be expressed and solved. A compiler-based approach is, therefore, the only reasonable option to the problem of optimizing local assembly operations.

Several studies have already tackled local assembly optimization in the context of automated code generation. In Olgaard and Wells [2010], it is shown how this technique can be leveraged to introduce domain-specific optimizations, which a user cannot be expected to write "by hand". Kirby et al. [2005] and Russell and Kelly [2013] have studied, instead, different optimization techniques based on a mathematical reformulation of the local assembly operations. The same problem has been addressed recently also for GPU architectures, for instance in Knepley and Terrel [2013], Klöckner et al. [2009], and Bana et al. [2014]. With our study, we make clear step forward by showing that different PDEs, on different platforms, require distinct sets of transformations if close-to-peak performance must be reached, and that low-level, domain-aware code transformations are essential to maximize instruction-level parallelism and register locality. As discussed in the following sections, our optimization strategy is quite different from those in previous work, although we reuse and leverage some of the ideas formulated in the available literature.

We present a novel structured approach to the optimization of automatically-generated local assembly kernels. We argue that for complex, realistic PDEs, peak performance can be achieved only by passing through a two-step optimization procedure: 1) expression rewriting, to minimize floating point operations, 2) and code specialization, to ensure effective register utilization and instruction-level parallelism, especially SIMD vectorization.

Expression rewriting consists of a framework capable of minimizing arithmetic intensity and optimize for register pressure. Our contribution is twofold:

- *Rewrite rules for assembly expressions.* The goal is to the reduce the computational intensity of local assembly kernels by rescheduling arithmetic operations based on a set of rewrite rules. These aggressively exploit associativity, distributivity, and commutativity of operators to expose loop-invariant sub-expressions and SIMD vectorization opportunities to the code specialization stage. While rewriting an assembly expression, domain knowledge is used in several ways, for example to avoid redundant computation.

- *An algorithm to deschedule useless operations.* Relying on symbolic execution, this algorithm restructures the code so as to skip useless arithmetic operations, for example multiplication by scalar quantities which are statically known to be zero. One problem is to transform the code while preserving code vectorizability, which is solved by resorting to domain-knowledge.

Code specialization's goal is to apply transformations to maximize the exploitation of the underlying platform's resources, e.g. SIMD lanes. We provide a number of contributions:

- *Padding and data alignment.* The small size of the loop nest (integration, test, and trial functions loops) require all of the involved arrays to be padded to a multiple of the vector register length so as to maximize the effectiveness of SIMD code. Data alignment can be enforced as a consequence of padding.
- *Vector-register Tiling.* Blocking at the level of vector registers, which we perform exploiting the specific memory access pattern of the assembly expressions (i.e. a domain-aware transformation), improves data locality beyond traditional unroll-and-jam optimizations. This is especially true for relatively high polynomial order (i.e. greater than 2) or when pre-multiplying functions are present.
- *Expression Splitting.* In certain assembly expressions the register pressure is significantly high: when the number of basis functions arrays (or, equivalently, temporaries introduced by loop-invariant code motion) and constants is large, spilling to L1 cache is a consequence for architectures with a relatively low number of logical registers (e.g. 16/32). We exploit sum's associativity to "split" the assembly expression into multiple sub-expressions, which are computed individually.
- *An algorithm to generate calls to BLAS routines.*
- *Autotuning.* We implement a model-driven, dynamic autotuner that transparently evaluates multiple sets of code transformations to determine the best optimization strategy for a given PDE. The main challenge here is to build, for a generic problem, a reasonably small search space that comprises most of the effective code variants.

Expression rewriting and code specialization have been implemented in a compiler, COFFEE[1], fully integrated with the Firedrake framework **?**. Besides separating the mathematical domain, captured by a domain-specific compiler at an higher level of abstraction, from the optimization process, COFFEE also aims to be platform-agnostic. The code transformations occur on an intermediate representation of the assembly operation, which is ultimately translated into platform-specific code. Domain knowledge is exploited in two ways: for simplifying the implementation of code transformations and to make them extremely effective. Domain knowledge is conveyed to COFFEE from the higher level through suitable annotations attached to the input. For example, when the input is in the form of an abstract syntax tree produced by the higher layer, specific nodes are decorated so as to drive the optimization process. Although COFFEE has been thought of as a multi-platform optimizing compiler, our performance evaluation so far has been restricted to standard CPU platforms only. We emphasize once more, however, that all of the transformations applicable would work on generic accelerators as well.

To demonstrate the effectiveness of our approach, we provide an extensive and unprecedented performance evaluation across a number of real-world PDEs of increasing complexity, including some based on complex hyperelasticity models. We characterize our problems by varying polynomial order of the employed function spaces and number of so called pre-multiplying functions. To clearly distinguish the improvements achieved by COFFEE, we will compare, for each examined PDE, four sets of code variants: 1) unoptimized code, i.e. a local assembly routine as returned from the domain-specific compiler; 2) code optimized by FEniCS, i.e. the work in Olgaard and Wells [2010]; 3) code optimized by expression rewriting and code specialization as described in this paper. Notable performance improvements of 3) over 1) and 2) are reported and discussed.

4.2 Preliminaries

In this section, the basic concepts sustaining the finite element method are summarized. The notation adopted in Olgaard and Wells [2010] and Russell and Kelly [2013] is followed. At the end of this section, the reader is expected to understand what local assembly represents and how an implementation can be derived starting from a mathematical specification of the finite element problem.

---

[1] COFFEE stands for COmpiler For Finit Element local assEmbly.

4.2.1 Overview of the Finite Element Method

We consider the weak formulation of a linear variational problem

$$\begin{aligned} &Find\ u\ \in U\ such\ that \\ &a(u,v) = L(v), \forall v \in V \end{aligned} \tag{4.1}$$

where a and L are called bilinear and linear form, respectively. The set of *trial* functions U and the set of *test* functions V are discrete function spaces. For simplicity, we assume $U = V$ and $\{\phi_i\}$ be the set of basis functions spanning U. The unknown solution u can be approximated as a linear combination of the basis functions $\{\phi_i\}$. From the solution of the following linear system it is possible to determine a set of coefficients to express u

$$A\mathbf{u} = b \tag{4.2}$$

in which A and b discretize a and L respectively:

$$\begin{aligned} A_{ij} &= a(\phi_i(x), \phi_j(x)) \\ b_i &= L(\phi_i(x)) \end{aligned} \tag{4.3}$$

The matrix A and the vector b are computed in the so called assembly phase. Then, in a subsequent phase, the linear system is solved, usually by means of an iterative method, and $\mathbf{u}$ is eventually evaluated.

We focus on the assembly phase, which is often characterized as a two-step procedure: *local* and *global* assembly. Optimizing the performance of local assembly is the subject of the research. Local assembly consists of computing the contributions that an element in the discretized domain provide to the approximated solution of the equation. Global assembly, on the other hand, is the process of suitably “inserting” such contributions in A and b.

4.2.2 Quadrature Representation for Finite Element Local Assembly

Without loss of generality, we illustrate local assembly in a concrete example, the evaluation of the local element matrix for a Laplacian operator. Consider the weighted Laplace equation

$$-\nabla \cdot (w \nabla u) = 0 \tag{4.4}$$

in which u is unknown, while w is prescribed. The bilinear form associated with the weak variational form of the equation is:

$$a(v, u) = \int_{\Omega} w \nabla v \cdot \nabla u \,\mathrm{d}x \tag{4.5}$$

The domain Ω of the equation is partitioned into a set of cells (elements) T such that $\bigcup T = \Omega$ and $\bigcap T = \emptyset$. By defining $\{\phi_i^K\}$ as the set of local basis functions spanning U on the element K, we can express the local element matrix as

$$A_{ij}^K = \int_K w \nabla \phi_i^K \cdot \nabla \phi_j^K \,\mathrm{d}x \tag{4.6}$$

The local element vector L can be determined in an analogous way starting from the linear form associated with the weak variational form of the equation.

Quadrature schemes are conveniently used to numerically evaluate A_{ij}^K. For convenience, a reference element K_0 and an affine mapping $F_K : K_0 \to K$ to any element $K \in T$ are introduced. This implies a change of variables from reference coordinates X_0 to real coordinates $x = F_K(X_0)$ is necessary any time a new element is evaluated. The numerical integration routine based on quadrature representation over an element K can be expressed as follows

$$A_{ij}^K = \sum_{q=1}^{N} \sum_{\alpha_3=1}^{n} \phi_{\alpha_3}(X^q) w_{\alpha_3} \sum_{\alpha_1=1}^{d} \sum_{\alpha_2=1}^{d} \sum_{\beta=1}^{d} \frac{\partial X_{\alpha_1}}{\partial x_\beta} \frac{\partial \phi_i^K(X^q)}{\partial X_{\alpha_1}} \frac{\partial X_{\alpha_2}}{\partial x_\beta} \frac{\partial \phi_j^K(X^q)}{\partial X_{\alpha_2}} det F_K' W^q \tag{4.7}$$

where N is the number of integration points, W^q the quadrature weight at the integration point X^q, d is the dimension of Ω, n the number of degrees of freedom associated to the local basis functions, and *det* the determinant of the Jacobian matrix used for the aforementioned change of coordinates.

In the next sections, we will often refer to the local element matrix evaluation, such as Equation 4.7 for the weighted Lapalce operator, as the *assembly expression* deriving from the weak variational problem.

4.2.3 Implementation of Quadrature-based Local Assembly

From Math to Code

We have explained that local assembly is the computation of contributions of a specific cell in the discretized domain to the linear system which yields the PDE solution. The process consists of numerically evaluating problem-specific integrals to produce a matrix and a vector (only the derivation of the matrix was shown in Section 4.2.2), whose sizes depend on the order of the method. This operation is applied to all cells in the discretized domain (mesh).

We consider again the weighted Laplace example of the previous section. A C-code implementation of Equation 4.7 is illustrated in Listing 1. The values at the various quadrature points of basis functions (ϕ) derivatives are tabulated in the FE0_D10 and FE0_D01 arrays. The summation along quadrature points q is implemented by the i loop, whereas the one along α_3 is represented by the r loop. In this example, we assume $d = 2$ (2D mesh), so the summations along α_1, α_2 and β have been straightforwardly expanded in the expression that evaluates the local element matrix A.

More complex assembly expressions, due to the employment of particular differential operators in the original PDE, are obviously possible. Intuitively, as the complexity of the PDE grows, the implementation of local assembly becomes increasingly more complicated. This fact is actually the real motivation behind reasearch in automated code generation techniques, such as those used by state-of-the-art frameworks like FEniCS and Firedrake. Automated code generation allows scientists to express the finite element specfication using a domain-specific language resembling mathematical notation, and obtain with no effort a semantically correct implementation of local assembly. The research in the present work is about making such an implementation also extremely effective, in terms of run-time performance, on standard CPU architectures.

The domain-specific language used by Firedrake and FEniCS to express finite element problems is the Unified Form Language (UFL) [Alnæs et al.,

LISTING 1: A possible implementation of Equation 4.7 assuming a 2D triangular mesh and polynomial order $p = 2$ Lagrange basis functions.

```
void weighted_laplace(double A[3][3], double **coordinates, double **w)
{
  // Compute Jacobian
  double J[4];
  compute_jacobian_triangle_2d(J, coordinates);

  // Compute Jacobian inverse and determinant
  double K[4];
  double detJ;
  compute_jacobian_inverse_triangle_2d(K, detJ, J);
  const double det = fabs(detJ);

  // Quadrature weights
  static const double W1[1] = 0.5;

  // Basis functions
  static const double FE0_D10[1][3] = {{ -0.999999999999999, ...}} ;
  static const double FE0_D01[1][3] = {{ -1.0, ...}} ;
  static const double FE0[1][3] = {{ 0.333333333333333, ...}} ;

  for (int i = 0; i ¡ 6; ++i)
  {
    double F0 = 0.0;
    for (int r = 0; r ¡ 3; ++r)
      F0 += (w[r][0] * FE0[i][r]);

    for (int j = 0; j ¡ 3; ++j)
      for (int k = 0; k ¡ 3; ++k)
        A[j][k] += (((((K[1]*FE0_D10[i][k])+(K[3]*FE0_D01[i][k])) *
                       ((K[1]*FE0_D10[i][j])+(K[3]*FE0_D01[i][j]))) +
                       (((K[0]*FE0_D10[i][k])+(K[2]*FE0_D01[i][k])) *
                       ((K[0]*FE0_D10[i][j])+(K[2]*FE0_D01[i][j]))))*det*W1[i]*F0);
  }
}
```

LISTING 2: UFL specification of the weighted Laplace equation for polynomial order $p = 2$ Lagrange basis functions.

```
// This is a Firedrake construct (not an UFL's) to instantiate a 2D mesh.
mesh = UnitSquareMesh(size, size)
// FunctionSpace also belongs to the Firedrake language
V = FunctionSpace(mesh, ”Lagrange”, 2)
u = TrialFunction(V)
v = TestFunction(V)
weight = Function(V).assign(value)
a = weight*dot(grad(v), grad(u))*dx
```

```
Input: element matrix (2D array, initialized to 0), coordinates (array),
       coefficients (array, e.g. velocity)
Output: element matrix (2D array)
- Compute Jacobian from coordinates
- Define basis functions
- Compute element matrix in an affine loop nest
```

Figure 4.1: Structure of a local assembly kernel

2014]. Listing 2 shows a possible UFL implementation for the weighted Laplace form. Note the resemblance of $a = weight$*... with Equation 4.6. A form compiler translates UFL code into the C code shown in Listing 1. We will describe these aspects carefully in Section 4.7; for the moment, this level of detail sufficies to open a discussion on how to optimize local assembly kernels arising from generic partial differential equations.

Other Examples and Motivations for Optimizations

The structure of a local assembly kernel can be generalized as in Figure 4.1. The inputs are a zero-initialized two dimensional array used to store the element matrix, the element's coordinates in the discretized domain, and coefficient fields, for instance indicating the values of velocity or pressure in the element. The output is the evaluated element matrix. The kernel body can be logically split into three parts:

1. Calculation of the Jacobian matrix, its determinant and its inverse required for the aforementioned change of coordinates from the reference element to the one being computed.

2. Definition of basis functions used to interpolate fields at the quadrature points in the element. The choice of basis functions is expressed in UFL directly by users. In the generated code, they are represented as global read-only two dimensional arrays (i.e., using `static const` in C) of double precision floats.

3. Evaluation of the element matrix in an affine loop nest, in which the integration is performed.

Table 4.1 shows the variable names we will use in the upcoming code snippets to refer to the various kernel objects.

Object name	Type	Variable name(s)
Determinant of the Jacobian matrix	double	det
Inverse of the Jacobian matrix	double	K1, K2, ...
Coordinates	double**	coords
Fields (coefficients)	double**	w
Numerical integration weights	double[]	W
Basis functions (and derivatives)	double[][]	X, Y, X1, ...
Element matrix	double[][]	A

Table 4.1: Type and variable names used in the various listings to identify local assembly objects.

LISTING 3: Local assembly implementation for a Helmholtz problem on a 2D mesh using polynomial order $p = 1$ Lagrange basis functions.

```
void helmholtz(double A[3][3], double **coords) {
  // K, det = Compute Jacobian (coords)

  static const double W[3] = {...}
  static const double X_D10[3][3] = {{...}}
  static const double X_D01[3][3] = {{...}}

  for (int i = 0; i<3; i++)
    for (int j = 0; j<3; j++)
      for (int k = 0; k<3; k++)
        A[j][k] += ((Y[i][k]*Y[i][j]+
          +((K1*X_D10[i][k]+K3*X_D01[i][k])*(K1*X_D10[i][j]+K3*X_D01[i][j]))+
          +((K0*X_D10[i][k]+K2*X_D01[i][k])*(K0*X_D10[i][j]+K2*X_D01[i][j])))*
          *det*W[i]);
}
```

The actual complexity of a local assembly kernel depends on the finite element problem being solved. In simpler cases, the loop nest is perfect, has short trip counts (in the range 3–15), and the computation reduces to a summation of a few products involving basis functions. An example is provided in Listing 3, which shows the assembly kernel for a Helmholtz problem using Lagrange basis functions on 2D elements with polynomial order $p = 1$. In other scenarios, for instance when solving the Burgers equation, the number of arrays involved in the computation of the element matrix can be much larger. The assembly code is given in Listing 4 and contains 14 unique arrays that are accessed, where the same array can be referenced multiple times within the same expression. This may also require the evaluation of constants in outer loops (called F in the code) to act as scaling factors of arrays. Trip counts grow proportionally to the order of the method and arrays may be block-sparse.

LISTING 4: Local assembly implementation for a Burgers problem on a 3D mesh using polynomial order $p = 1$ Lagrange basis functions.

```
void burgers(double A[12][12], double **coords, double **w) {
  // K, det = Compute Jacobian (coords)

  static const double W[5] = {...}
  static const double X1_D001[5][12] = {{...}}
  static const double X2_D001[5][12] = {{...}}
  //11 other basis functions definitions.
  ...
  for (int i = 0; i<5; i++) {
    double F0 = 0.0;
    //10 other declarations (F1, F2,...)
    ...
    for (int r = 0; r<12; r++) {
      F0 += (w[r][0]*X1_D100[i][r]);
      //10 analogous statements (F1, F2, ...)
      ...
    }
    for (int j = 0; j<12; j++)
      for (int k = 0; k<12; k++)
        A[j][k] += (..(K5*F9)+(K8*F10))*Y1[i][j])+
          +(((K0*X1_D100[i][k])+(K3*X1_D010[i][k])+(K6*X1_D001[i][k]))*Y2[i][j]))*F11)+
      +(..((K2*X2_D100[i][k])+...+(K8*X2_D001[i][k]))*((K2*X2_D100[i][j])+...+(K8*X2_D001[i][j]))..)+
          + <roughly a hundred sum/muls go here>)..)*
          *det*W[i]);
  }
}
```

In general, the variations in the structure of mathematical expressions and in loop trip counts (although typically limited to the order of tens of iterations) that different equations show, render the optimization process challenging, requiring distinct sets of transformations to bring performance closest to the machine peak. For example, the Burgers problem, given the large number of arrays accessed, suffers from high register pressure, whereas the Helmholtz equation does not. Moreover, arrays in Burgers are block-sparse due to the use of vector-valued basis functions (we will elaborate on this in the next sections). These few aspects (we could actually find more) already intuitively suggests that the two problems require a different treatment, based on an in-depth analysis of both data and iteration spaces. Furthermore, domain knowledge enables transformations that a general-purpose compiler could not apply, making the optimization space even larger. In this context, our goal is to understand the relationship between distinct code transformations, their impact on cross-loop arithmetic intensity, and to what extent their composability is effective in a wide class of real-world equations and architectures.

We also note that despite the infinite variety of assembly kernels that frameworks like FEniCS and Firedrake can generate, it is still possible to identify common domain-specific traits that are potentially exploitable for our optimization strategy. These include: 1) memory accesses along the three loop dimensions are always unit stride; 2) the `j` and `k` loops are interchangeable, whereas interchanges involving the i loop require precomputation of values (e.g. the F values in Burgers) and introduction of temporary arrays (explained next); 3) depending on the problem being solved, the `j` and `k` loops could iterate along the same iteration space; 4) most of the sub-expressions on the right hand side of the element matrix computation depend on just two loops (either `i-j` or `i-k`). In the following sections we show how to exploit these observations to define a set of systematic, composable optimizations.

4.3 Overview of the Optimization Strategy

To generate high performance implementation of local assembly kernels, assembly expressions must be optimized with regards to three interrelated aspects:

- arithmetic intensity
- instruction-level parallelism
- data locality

Three conceptually distinct kind of transformations can be individuated, which we refer to as *expression rewriting*, *code specialization*, and *general-purpose optimizations*. Expression rewriting mainly targets arithmetic intensity by transforming the assembly expression (and its enclosing loop nest) so as to minimize the number of floating point operations required to evaluate the local element matrix. Code specialization is tailored to optimizing for instruction-level parallelism, particularly SIMD vectorization, and data (register) locality. Both classes of transformations are inspired by the inherent structure of local assembly code and make use of domain knowledge: as elaborated in the next sections, these two aspects are the core motivations for which standard general-purpose compilers fail at maximizing the performance of local assembly kernels. The third class of transformations is about general-purpose optimizations; that is, generic, well-known techniques for improving code performance, such as loop unrolling or loop interchange, that for some reasons are not applied by the compiler generating machine code, but potentially useful in certain equations.

In Section 4.7, it is also explained how these classes of transformations must be applied following a specific order. Also, some effort was invested to ensure that optimizations at stage i (e.g. expression rewriting) would not break any further optimization opportunity at stage $i+1$ (code specialization). In the following, we present the various classes of transformations following the order in which they are expected to be applied to a local assembly kernel.

4.4 Expression Rewriting

Expression rewriting is about exploiting properties of operators such as associativity, distributivity, and commutativity, to minimize arithmetic intensity, expose code vectorization opportunities, and optimize the register pressure in the various levels of the assembly loop nest. There are many possibilities of rewriting an expression, so the transformation space can be quite large. Firstly, in Sections 4.4.1–4.4.4, several ways of manipulating an assembly expression and their potential impact on the computational cost are described. Then, in Section 4.4.5, a simple yet systematic way of rewriting an expression based on a set of formal rewrite rules, which is the key to effectively explore the expression's transformation space, is formalized.

The second objective of expression rewriting (Section 4.4.6) consists of restructuring the iteration space so as to avoid arithmetic operations over zero-valued columns in block-sparse basis functions arrays. Zero-valued columns arise, for example, when taking derivatives on a reference element or when using mixed (vector-valued) elements. This problem was tackled in Olgaard and Wells [2010], but the proposed solution, which makes use of indirection arrays in the generated local assembly code, breaks most of the optimizations applicable by code specialization, including the fundamental SIMD vectorization. The contribution of the present research is a novel domain-aware approach based on symbolic execution that avoids indirection arrays.

```
for (int i=0; i<I; ++i) {
  // Coefficient evaluation at point i
  for (int r=0; r<T; ++r) {
    f0 += ...; f1 += ...; ...;
  }
  // Test functions
  for (int j=0; j<T; ++j)
  {
    // Trial functions
    for (int k=0; k<T; ++k)
    {
      M[j][k] += (((A[i][k]/c)*A[i][j])+
                  ((a*f0*A[i][j]+b*f1*B[i][j])*A[i][k]*g)+
                  (((d*D[i][k]+e*E[i][k])*A[i][j])*f))*det*W[i]
    }
  }
}
```

Figure 4.2: Original (simplified) code

4.4.1 Generalized Loop-invariant Code Motion

Consider the local element matrix computation in Figure 4.4.1, which is an excerpt from the Burgers problem shown in Listing 4. The assembly expression, produced by the FEniCS and Firedrake's form compiler, has been deliberately simplified, and code details have been omitted for brevity and readability. In practice, as already emphasized, assembly expressions can be much more complex depending on the differential operators employed in the variational form; however, this example is representative enough for highlighting patterns that are common in a large class of equations.

A first glimpse of the code suggests that the `a*f0*A[i][j]+b*f1*B[i][j]` sub-expression is invariant with respect to the innermost (trial functions) loop `k`, so it can be hoisted at the level of the outer loop `j` to avoid redundant computation. This is indeed a standard compiler transformation, supported by any available compilers, so, in principle, there should be no need to transform the source code explicitly. With a closer look we notice that the sub-expression `d*D[i][k]+e*E[i][k]` is also invariant, although, this time, with respect to the outer (test functions) loop `j`. Available compilers (e.g. *GNU's* and *Intel's*) limit the search for code motion opportunities to the innermost loop of a given nest. Moreover, the hoisted code is scalar

```
for (int i=0; i<I; ++i) {
  for (int r=0; r<T; ++r) {
    f0 += ...; f1 += ...; ...;
  }
  double T1[T], T2[T];
  for (int r=0; r<T; ++r) {
    T1[r] = ((f0*a*A[i][r])+(f1*b*B[i][r]));
    T2[r] = ((d*D[i][r])+(e*E[i][r]));
  }
  for (int j=0; j<T; ++j) {
    for (int k=0; k<T; ++k) {
      M[j][k] += (((A[i][k]/c)*A[i][j])+
                  (T1[j]*A[i][k]*g)+
                  (T2[k]*A[i][j]*f))*det*W[i];
    }
  }
}
```

Figure 4.3: Invariant code

and therefore not subjected to SIMD auto-vectorization. In other words, these general-purpose compilers lack performance models to determine (i) the optimal place where to hoist an expression and (ii) the potential gain and overhead (due to the need for extra temporary memory) of vectorization. These are notable limitations for local assembly kernels.

We work around these limitations with source-level loop-invariant code motion. In particular, we pre-compute all values that an invariant sub-expression assumes along its fastest varying dimension. This is implemented by introducing a temporary array per invariant sub-expression and by adding a new loop to the nest. At the price of extra memory for storing temporaries, the gain is that lifted terms can be auto-vectorized as part of an inner loop. Given the short trip counts of our loops, it is important to achieve auto-vectorization of hoisted terms in order to minimize the percentage of scalar instructions, which could otherwise be significant. It is also worth noting that, in some problems, for instance Helmholtz in Listing 3, invariant sub-expressions along `j` are identical to those along `k`, and both loops iterate over the same iteration space, as anticipated in Section **??**. In these cases, we safely avoid redundant pre-computation. The resulting code for the running Burgers example is shown in Figure 4.4.1.

```
for (int i=0; i<I; ++i) {
  for (int r=0; r<T; ++r) {
    f0 += ...; f1 += ...; ...;
  }
  double T1[T], T2[T];
  for (int r=0; r<T; ++r) {
    T1[r] = ((f0*a*A[i][r])+(f1*b*B[i][r]));
    T2[r] = ((d*D[i][r])+(e*E[i][r]));
    T3[r] = ((A[i][r]/c)+T2[r]*f);
  }
  for (int j=0; j<T; ++j) {
    for (int k=0; k<T; ++k) {
      M[j][k] += ((T1[j]*A[i][k]*g)+
                  (T3[k]*A[i][j]))*det*W[i];
    }
  }
}
```

Figure 4.4: Factorized code

In the following, we refer to this series of transformations as *generalized loop-invariant code motion*. We will show that this optimization is crucial when optimizing non-trivial assembly expressions, allowing to achieve performance improvements over the original code larger than 3×.

4.4.2 Terms Factorization

After generalized loop-invariant code motion has been applied, some assembly expressions can still “hide” opportunities for code hoisting. By examining again the code in Figure 4.4.1, we notice that the basis function array `A` iterating along the `[i,j]` loops appears twice in the expression. By expanding the products in which `A` is accessed and by applying sum commutativity, terms can be factorized. This has two effects: firstly, it reduces the number of arithmetic operations performed; secondly, and most importantly, it exposes a new sub-expression `A[i][k]/c+T2[k]*f` invariant with respect to loop `j`. Consequently, hoisting can be performed, resulting in the code in Figure 4.4.2. In general, exposing factorization opportunities requires traversing the whole expression tree, and then expanding and moving terms. It also needs heuristics to select a factorization strategy: there may be different opportunities of reorganizing sub-expressions, and,

```
for (int i=0; i<I; ++i) {
  for (int r=0; r<T; ++r) {
    f0[i] += ...; f1[i] += ...; ...;
  }
  k0[i] = f0[i]*a; k1[i] = f1*b;
}
for (int i=0; i<I; ++i) {
  double T1[T], T2[T];
  for (int r=0; r<T; ++r) {
    T1[r] = ((k0[i]*A[i][r])+(k1[i]*B[i][r]));
    T2[r] = ((d*D[i][r])+(e*E[i][r]));
    T3[r] = ((A[i][r]/c)+T2[r]*f);
  }
  for (int j=0; j<T; ++j) {
    for (int k=0; k<T; ++k) {
      M[j][k] += ((T1[j]*A[i][k]*g)+
                  (T3[k]*A[i][j]))*det*W[i];
    }
  }
}
```

Figure 4.5: Scalar-expanded code

in our case, the best is the one that maximizes the invariant code eventually disclosed. We will discuss this aspect formally in Section 4.4.5.

4.4.3 Precomputation of Invariant Terms

We note that integration-dependent expressions are inherently executed as scalar code. For example, the `f0*a` and `f1*b` products in Figure **??** depend on the loop along quadrature points; these operations are performed in a non-vectorized way at every `i` iteration. This is not a big issue in our running example, in which the scalar computation represents a small fraction of the total, but it becomes a concrete problem in complicated forms, like those at the heart of hyperelasticity (which will be part of our performance evaluation). In such forms, the amount of computation independent of both test and trial functions loops is so large that it has a significant impact on the run-time, despite being executed only $O(I)$ times (with I number of quadrature points). We have therefore implemented an algorithm to move and scalar-expand integration-dependent expressions, which leads to codes as in Figure 4.4.3.

4.4.4 Expanding Sub-expressions

Expression rewriting also aims at minimizing register pressure in the assembly loop nest. Once the code has been optimized for arithmetic intensity, it is important to think about how the transformations impacted register allocation. Assume the local assembly kernel is executed on a state-of-the-art CPU architecture having 16 logical registers, e.g. an Intel Haswell. Each value appearing in the expression is loaded and kept in a register as long as possible. In Figure 4.4.3, for instance, the scalar value `g` is loaded once, whereas the term `det*W[i]` is precomputed and loaded in a register at every `i` iteration. This implies that at every iteration of the `[j,k]` loop nest, 12% of the available registers are spent just to store values independent of test and trial functions loops. In more complicated expressions, the percentage of registers destined to store such constant terms can be even higher. Registers are, however, a precious resource, especially when evaluating compute-intensive expressions. The smaller is the number of available free registers, the worse is the instruction-level parallelism achieved: for example, a shortage of registers can increase the pressure on the L1 cache (i.e. it can worsen data locality), or it may prevent the effective application of standard transformations, e.g. loop unrolling. We aim at relieving this problem by suitably expanding terms and introducing, where necessary, additional temporary values. We illustrate this in the following example.

Consider a variant of the Burgers local assembly kernel, shown in Figure 4.6(a). This is again a representative, simplified example. We can easily distribute `det*W[i]` over the three operands on the left-hand side of the multiplication, and then absorb it in the pre-computation of the invariant sub-expression stored in `T1`, resulting in code as in Figure 4.6(b). Freeing the register destined to the constant `g` is less straightforward: we cannot absorb it in `T1` as we did with `det*W[i]` since `T1` is also accessed in the `T1[j]*A[i][k]` sub-expression. The solution is to add another temporary as in Figure 4.6(c). Generalizing, this is a problem of data dependencies: to solve it, we use a dependency graph in which we add a direct edge from identifier `A` to identifier `B` to denote that the evaluation of `B` depends on `A`. The dependency graph is initially empty, and is updated every time a new temporary is created by either loop-invariant code motion or expansion of terms. The dependency graph is then queried to understand when expan-

```
for (int i=0; i<I; ++i) {
  double T1[T];
  for (int r=0; r<T; ++r) {
    T1[r] = ((a*A[i][r])+(b*B[i][r]));
  }
  for (int j=0; j<T; ++j) {
    for (int k=0; k<T; ++k) {
      M[j][k] += (((T1[j]*A[i][k])+(T1[k]*A[i][j]))*g+
                    +(T1[j]*A[i][k]))*det*W[i];
    }
  }
}
```

(a) Original (simplified) code

```
for (int i=0; i<I; ++i) {
  double T1[T];
  for (int r=0; r<T; ++r) {
    T1[r] = ((a*A[i][r])+(b*B[i][r]))*det*W[i];
  }
  for (int j=0; j<T; ++j) {
    for (int k=0; k<T; ++k) {
     M[j][k] += (T1[j]*A[i][k]+T1[k]*A[i][j])*g+(T1[j]*A[i][k]);
    }
  }
}
```

(b) The code after expansion of `det*W[i]`

```
for (int i=0; i<I; ++i) {
  double T1[T], T2[T];
  for (int r=0; r<T; ++r) {
    T1[r] = ((a*A[i][r])+(b*B[i][r]))*det*W[i];
    T2[r] = T1[r]*g;
  }
  for (int j=0; j<T; ++j) {
    for (int k=0; k<T; ++k) {
     M[j][k] += (T2[j]*A[i][k])+(T2[k]*A[i][j])+(T1[j]*A[i][k]);
    }
  }
}
```

(c) The code after expansion of `g`. Note the need to introduce a new temporary array.

Figure 4.6: Expansion of terms to improve register pressure in a local assembly kernel

sion can be performed without resorting to new temporary values. This aspect is formalized in the next section.

4.4.5 Rewrite Rules for Assembly Expressions

In general, assembly expressions produced by automated code generation can be much more complex than those we have used as examples, with dozens of terms involved (basis function arrays, derivatives, coefficients, ...) and hundreds of (nested) arithmetic operations. Our goal is to establish a portable, unified, platform-independent, and systematic way of reducing the computational strength of an expression exploiting the intuitions described in the previous section. This *expression rewriting* should be simple; definitely, it must be robust to be integrated in an optimizing domain-specific compiler capable of supporting real problems, such as COFFEE. In other words, we look for an algorithm capable of transforming a plain assembly expression by applying 1) generalized loop-invariant code motion, 2) non-trivial factorization and re-association of sub-expressions, and 3) expansion of terms; after that, it should perform 4) scalar-expansion of integration-dependent terms to achieve full vectorization of the assembly code.

We centre such an algorithm around a set of rewrite rules. These rules drive the transformation of an expression, prescribe where invariant sub-expressions will be moved (i.e. at what level in the loop nest), and track the propagation of data dependencies. When applying a rule, the state of the loop nest must be updated to reflect, for example, the use of a new temporary and new data dependencies. We define the state of a loop nest as $L = (\sigma, G)$, where $G = (V, E)$ represents the dependency graph, while $\sigma : Inv \rightarrow S$ maps invariant sub-expressions to identifiers of temporary arrays. The notation σ_i refers to invariants hoisted at the level of loop i. We also introduce the *conditional hoister* operator [] on σ such that

$$\sigma[{}^{v}\!/\!_{x}] = \begin{cases} \sigma(x) & \text{if } x \in Inv\text{; } v \text{ is ignored} \\ v & \text{if } x \notin Inv\text{; } \sigma(\text{x}) = \text{v} \end{cases}$$

That is, if the invariant expression `x` has already been hoisted, [] returns the temporary identifier hosting its value; otherwise, `x` is hoisted and a new temporary v is created. There is a special case when $v =\perp$, used for

Rule	*Precondition*
$[a_i \cdot b_j]_{(\sigma,G)} \rightarrow [a_i \cdot b_j]_{(\sigma,G)}$	
$[(a_i + b_j) \cdot \alpha]_{(\sigma,G)} \rightarrow [(a_i \cdot \alpha + b_j \cdot \alpha)]_{(\sigma,G)}$	
$[a_i \cdot b_j + a_i \cdot c_j]_{(\sigma,G)} \rightarrow [(a_i \cdot (b_j + c_j)]_{(\sigma,G)}$	
$[a_i + b_i]_{(\sigma,G)} \rightarrow [t_i]_{(\sigma',G')}$	$t_i = \sigma_{i_0}[t'_i/a_i + b_i], G' = (V \cup t_i, E \cup \{(t_i, a_i), (t_i, b_i)\})$
$[(a_i \cdot b_j) \cdot \alpha]_{(\sigma,G)} \rightarrow [t_i \cdot b_j]_{(\sigma',G')}$	$\sharp(b_j) > \sharp(a_i), t_i = \sigma_{i_0}[\sigma[\,\perp/a_i]/a_i \cdot \alpha], a_i \notin in(G),$ $G' = (V \cup t_i, E \cup \{(t_i, a_i), (t_i, \alpha)\})$
$[(a_i \cdot b_j) \cdot \alpha]_{(\sigma,G)} \rightarrow [t_i \cdot b_j]_{(\sigma',G')}$	$\sharp(b_j) > \sharp(a_i), t_i = \sigma_{i_0}[t'_i/a_i \cdot \alpha], a_i \in in(G),$ $G' = (V \cup t_i, E \cup \{(t_i, a_i), (t_i, \alpha)\})$

Figure 4.7: Rewrite rules driving the rewriting process of an assembly expression.

conditional deletion of entries in σ. Specifically

$$\sigma[\,\perp/x] = \begin{cases} \sigma(x) & \text{if } x \in Inv;\ \sigma = \sigma \setminus (x, \sigma(x)) \\ v & \text{if } x \notin Inv;\ v \notin S \end{cases}$$

In other words, the invariant sub-expression `x` is removed and the temporary identifier that was hosting its value is returned if `x` had been previously hoisted; otherwise, a fresh identifier v is returned. This is useful to express updates of hoisted invariant sub-expressions when expanding terms.

In the following, a generic (sub-)expression is represented with Roman letters `a, b, ...`; constant terms are considered a special case, so Greek letters α`,` β`, ...` are used instead. The iteration vector $i = [i_0, i_1, ...]$ is the ordered sequence of the indices of the loops enclosing an (sub-)expression. We will refer to i_0 as the outermost enclosing loop. The notation a_i, therefore, indicates that the expression a assumes distinct values while iterating along the loops in i; its outermost loop is silently assumed to be i_0.

Rewrite rules for expression rewriting are provided in Figure 4.7; obvious rules are omitted for brevity. The Expression Rewriter applies the rules while performing a depth-first traversal of the assembly expression tree. Given an arithmetic operation between two sub-expressions (i.e. a node in the expression tree), we first need to find an applicable rule. There cannot be ambiguities: only one rule can be matched. If the preconditions of the rule are satisfied, the corresponding transformation is performed; otherwise, no rewriting is performed, and the traversal proceeds. As examples, it is possible to instantiate the rules in the codes shown in Figures 4.4.1 and 4.6(a); eventually, the optimized codes in Figures 4.4.2 and 4.6(c) are obtained, respectively.

To what extent should rewrite rules be applied is a question that cannot be answered in general. In some problems, a full rewrite of the expression may be the best option; in other cases, on the other hand, an aggressive expansion of terms, for example, may lead to high register pressure in the loops computing invariant terms, worsening the performance. In Section 4.5 how to leverage code specialization to select a suitable rewriting strategy for the problem at hand is explained.

4.4.6 Avoiding Iteration over Zero-valued Blocks by Symbolic Execution

Skipping arithmetic operations over blocks of zero-valued entries in basis functions arrays is the second goal of expression rewriting. Zero-valued columns arise, for example, when taking derivatives on a reference element and when employing mixed elements. In Olgaard and Wells [2010], a technique to avoid operations on zero-valued columns based on the use of indirection arrays was described (e.g. `A[B[i]]`, where `A` is a tabulated basis function and `B` a map from loop iterations to non-zero columns in `A`) and implemented in the FEniCS framework. The approach proposed in this section will be evaluated and compared to this pioneering work. Essentially, the strategy avoids indirection arrays in the generated code, which otherwise would break the optimizations applicable by code specialization, including SIMD vectorization.

In Figure 4.8(a), an enriched version of the Burgers excerpt in Figure **??** is illustrated. The code is instantiated for the specific case of polynomial order $p = 1$ Lagrange basis functions on a 2D mesh. The array `D` represents a derivative of a basis function tabulated at the various quadrature points. There are four zero-valued columns. Any multiplications or additions along these columns could (should) be skipped to avoid irrelevant floating point operations. The solution adopted in Olgaard and Wells [2010] is not to

```
void burgers(double M[6][6],
             double **coordinates,
             double **coeff0) {
  // Calculate determinant of jacobian
  // (det) using the coordinates field

  // Define tabulation of basis functions
  // (and their derivatives) arrays
  ...
  @coffee mfs[3, 5]
  static const double D[6][6] =
     {{0.0, 0.0, 0.0, -1.0, 0.0, 1.0},
      {0.0, 0.0, 0.0, -1.0, 0.0, 1.0},
      {0.0, 0.0, 0.0, -1.0, 0.0, 1.0},
      {0.0, 0.0, 0.0, -1.0, 0.0, 1.0},
      {0.0, 0.0, 0.0, -1.0, 0.0, 1.0},
      {0.0, 0.0, 0.0, -1.0, 0.0, 1.0}};
  ...
  for (int i=0; i<6; ++i) {
    ...
    double T1[6], T2[6];
    for (int r=0; r<6; ++r) {
      T1[r] = a*A[i][r]+b*B[i][r];
      T2[r] = d*D[i][k]+e*E[i][k];
    }
    for (int j=0; j<6; ++j)
      for (int k=0; k<6; ++k)
        M[j][k] += (T1[j], T2[k], A[i][j], ...)
  }
}
```

(a) Original code. Note the annotation over the definition of the tabulated basis function `D`, which is used to identify the presence of zero-valued columns

```
void burgers(double M[6][6],
             double **coordinates,
             double **coeff0) {
  // Calculate determinant of jacobian (det)
  // using the coordinates field

  // Define tabulation of basis functions
  // (and their derivatives) arrays
  ...
  @coffee mfs[3, 5]
  static const double D[6][6] =
     {{0.0, 0.0, 0.0, -1.0, 0.0, 1.0},
      {0.0, 0.0, 0.0, -1.0, 0.0, 1.0},
      {0.0, 0.0, 0.0, -1.0, 0.0, 1.0},
      {0.0, 0.0, 0.0, -1.0, 0.0, 1.0},
      {0.0, 0.0, 0.0, -1.0, 0.0, 1.0},
      {0.0, 0.0, 0.0, -1.0, 0.0, 1.0}};
  ...
  for (int i=0; i<6; ++i) {
    ...
    double T1[6], T2[6];
    for (int r=0; r<6; ++r) {
      T1[r] = a*A[i][r]+b*B[i][r];
      T2[r] = d*D[i][k]+e*E[i][k];
    }
    for (int j0=0; j0<3; ++j0)
      for (int k0=0; k0<3; ++k0)
        M[j0+3][k0+3] += f(T1[j0+3], A[i][k0+3], ...);
    for (int j1=0; j1<3; ++j1)
      for (int k1=0; k1<6; ++k1)
        M[j1][k1] += g(T2[k1], A[i][j1+3], ...);
  }
}
```

(b) The code after symbolic execution took place

Figure 4.8: Simplified excerpt of local assembly code from a Burgers form using vector-valued basis functions, before and after symbolic execution is performed to rewrite the iteration space

generate the zero-valued columns (i.e. to generate a dense 6×2 array), to reduce the size of the iteration space over test and trial functions (from 6 to 2), and to use an indirection array (e.g. $ind = \{3, 5\}$) to update the right entries in the element tensor A. This prevents, among the various optimizations, effective SIMD vectorization, because memory loads and store would eventually reference non-contiguous locations.

The new strategy exploits domain knowledge and makes use of symbolic execution. We discern the origin of zero-valued columns: for example, those due to taking derivatives on the reference element from those inherent to using mixed (vector) elements. In the running Burgers example, the use of vector-valued basis functions require the generation of a zero-valued block (columns 0, 1, 2 in the array `D`) to correctly evaluate the local element matrix while iterating along the space of test and trial functions. The two key observations are that: (i) the number of zero-valued columns caused by using vector function spaces is, often, much larger then that due to derivatives, and (ii) such columns are contiguous in memory. Based on this observation, we aim to avoid iteration only along the block of zero-valued columns induced by mixed (vector) elements.

The goal is achieved by means of symbolic execution. The algorithm expects some indication about the location of the zero-valued columns induced by mixed (vector) function spaces, for each tabulated basis function. This information comes either in the form of code annotation, if the input to COFFEE were provided as pure C (as shown in Figure 4.8(a)), or by suitably decorating basis functions nodes, if the input were an abstract syntax tree. Then, the rewrite rules are applied and each statement is executed symbolically. For example, consider the assignment `T2[r] = d*D[i][k]+e*E[i][k]` in Figure 4.8(a). Array `D` has non-zero-valued columns in the range $NZ_D = [3, 5]$; we also assume array `E` has non-zero-valued columns in the range $NZ_E = [0, 2]$. Multiplications by scalar values do not affect the propagation of non-zero-valued columns. On the other hand, when symbolically executing the sum of the two operands `d*D[i][k]` and `e*E[i][k]`, we track that the target identifier `T2` will have non-zero-valued columns in the range $NZ_E \| NZ_D = [0, 5]$. Eventually, exploiting the NZ information computed and associated with each identifier, we split the original assembly expression into multiple sets of sub-expressions, each set characterized by the same range of non-zero-valued columns. In our example, assuming that $NZ_{T1} = [3, 5]$ and $NZ_A = [3, 5]$, there are two of such sets, which leads to the generation of two distinct iteration spaces (one for each set), as in Figure 4.8(b).

4.5 Code Specialization

Code specialization's goal is architecture-specific optimization for instruction-level parallelism and register locality.

4.5.1 Padding and Data Alignment

The absence of stencils renders the local element matrix computation easily auto-vectorizable by a general-purpose compiler. Nevertheless, auto-vectorization is not efficient if data are not aligned to cache-line boundaries and if the length of the innermost loop is not a multiple of the vector length `VL`, especially when the loops are small as in local assembly.

Data alignment is enforced in two steps. Firstly, all arrays are allocated to addresses that are multiples of `VL`. Then, two dimensional arrays are padded by rounding the number of columns to the nearest multiple of `VL`. For instance, assume the original size of a basis function array is 3×3 and `VL` = 4 (e.g. AVX processor, with 256-bit long vector registers and 64-bit double-precision floats). In this case, a padded version of the array will have size 3×4. The compiler is explicitly told about data alignment using suitable pragmas; for example, in the case of the Intel compiler, the annotation `#pragma vector aligned` is added before the loop (as shown in later figures) to inform that all of the memory accesses in the loop body will be properly aligned. This allows the compiler to issue aligned load and store instructions, which are notably faster than unaligned ones.

Padding of all two dimensional arrays involved in the evaluation of the element matrix also allows to safely round the loop trip count to the nearest multiple of `VL`. This avoids the introduction of a remainder (scalar) loop from the compiler, which would render vectorization less efficient. These extra iterations only write to the padded region of the element matrix, and therefore have no side effects on the final result.

Example

Consider again the code in Figure 4.8(b). The arrays in the loop nest `[j1,k1]` can be padded and the right bound of loop `k1` can be safely increased to 8: eventually, values computed in the region `M[0:3][6:8]` will be discarded. Then, by explicitly aligning arrays and using suitable pragmas (e.g. `#pragma simd` for the Intel compiler), effective SIMD auto-vectorization can be obtained for this loop nest.

There are some complications in the case of loops `[j0,k0]`. Here, increasing the loop bound to 4 is still safe, assuming that both `T1` and `A` are padded with zero-valued entries, but it has no effect: the starting addresses of the load instructions would be `T1[3]` and `A[i][3]`, which are not aligned. One solution is to start iterating from the closest index that would ensure data alignment: in this specific case, $k0 = 0$. However, this would imply losing (partially in general, totally for this loop nest) the effect of the zero-avoidance transformation. Another possibility is to attain to non-aligned accesses. Which of the two strategy is better cannot be established a-priori, nor is east to formalize a cost model that helps discerning between them; an autotuning system, as described in Section 4.7.4, can be used to answer this question.

Note that when the extra iterations end up accessing non-zero entries in the local element matrix `M` there is no way of recovering data alignment.

4.5.2 Expression Splitting

In complex kernels, like Burgers in Listing 4, and on certain architectures, achieving effective register allocation can be challenging. If the number of variables independent of the innermost-loop dimension is close to or greater than the number of available CPU registers, poor register reuse is likely. This usually happens when the number of basis function arrays, temporaries introduced by generalized loop-invariant code motion, and problem constants is large. For example, applying loop-invariant code motion to Burgers on a 3D mesh requires 24 temporaries for the `ijk` loop order. This can make hoisting of the invariant loads out of the `k` loop inefficient on architectures with a relatively low number of registers. One potential solution to this problem consists of suitably "splitting" the computation of the element matrix A into multiple sub-expressions; an example, for the Burgers

```
for (int i=0; i<I; ++i) {
  double T1[T], T2[T];
  for (int r=0; r<T; ++r) {
    T1[r] = ((a*A[i][r])+(b*B[i][r]))*det*W[i];
    T2[r] = T1[r]*g;
  }
  for (int j=0; j<T; ++j) {
    for (int k=0; k<T; ++k) {
     M[j][k] += (T2[j]*A[i][k])+(T2[k]*A[i][j]);
    }
  }
  for (int j=0; j<T; ++j) {
    for (int k=0; k<T; ++k) {
     M[j][k] += (T1[j]*A[i][k]);
    }
  }
}
```

Figure 4.9: Local assembly code for the Burgers problem in Figure 4.6(c) after the application of *split*. In this example, the split factor is 2.

problem, is given in Listing 4.5.2. The transformation can be regarded as a special case of classic loop fission, in which associativity of the sum is exploited to distribute the expression across multiple loops. To the best of our knowledge, expression splitting is not supported by available compilers.

Splitting an expression (henceforth *split*) has, however, several drawbacks. Firstly, it increases the number of accesses to A proportionally to the "split factor", which is the number of sub-expressions produced. Also, depending on how the split is executed, it can lead to redundant computation. For example, the number of times the product $det * W3[i]$ is performed is proportional to the number of sub-expressions, as shown in the code snippet. Further, it increases loop overhead, for example through additional branch instructions. Finally, it might affect register locality: for instance, the same array could be accessed in different sub-expressions, requiring a proportional number of loads be performed. This is not the case for the Helmholtz example. Nevertheless, as shown in Section 4.8.1, the performance gain from improved register reuse along inner dimensions can still be greater, especially if the split factor and the splitting itself use heuristics to minimize the aforementioned issues.

```
void burgers(double M[6][6], double **coordinates) {
  // Calculate determinant of jacobian (det) using the coordinates
      field
  // Define tabulation of basis functions (and their derivatives)
      arrays
  ...

  for (int i=0; i<6; ++i) {
    ...
    double T1[6], T2[6];
    for (int r=0; r<6; ++r) {
      T1[r] = a*A[i][r]+b*B[i][r];
      T2[r] = d*D[i][k]+e*E[i][k];
    }

    for (int j=0; j<4; ++j) {
      for (int k=0; k<8; ++k) {
        // "load" and "set" intrinsics
        // Compute M[0,0], M[1,1], M[2,2], M[3,3]
        // One "permute_pd" intrinsic per k-loop "load"
        // Compute M[0,1], M[1,0], M[2,3], M[3,2]
        // One "permute2f128_pd" intrinsic per k-loop load
        ...
      }
    }

    // Reminder loop
    for (int j=4; j<6; ++j) {
      for (int k=0; k<6; ++k) {
        M[j][k] += (T1[j], T2[k], A[i][j], ...)
      }
    }

    // Restore the storage layout:\\
    for (int j = 0; j<4; j+=4) {
      _m256d r0, r1, r2, r3, r4, r5, r6, r7;
      for (int k = 0; k<8; k+=4) {
        r0 = _mm256_load_pd (&A[j+0][k]);
        // Load A[j+1][k], A[j+2][k], A[j+3][k]
        r4 = _mm256_unpackhi_pd (r1, r0);
        r5 = _mm256_unpacklo_pd (r0, r1);
        r6 = _mm256_unpackhi_pd (r2, r3);
        r7 = _mm256_unpacklo_pd (r3, r2);
        r0 = _mm256_permute2f128_pd (r5, r7, 32);
        r1 = _mm256_permute2f128_pd (r4, r6, 32);
        r2 = _mm256_permute2f128_pd (r7, r5, 49);
        r3 = _mm256_permute2f128_pd (r6, r4, 49);
        mm256_store_pd (&A[j+0][k], r0);
        // Store M[j+1][k], M[j+2][k], M[j+3][k]
      }
    }
  }
}
```

Figure 4.10: Local assembly code for the Burgers problem in Figure 4.8(a) after the application of vector-register tiling (outer-product vectorization). In this example, the unroll-and-jam factor is 1.

4.5.3 Model-driven Vector-register Tiling

One notable problem of assembly kernels concerns register allocation and register locality. The critical situation occurs when loop trip counts and the variables accessed are such that the vector-register pressure is high. Since the kernel's working set fits the L1 cache, it is particularly important to optimize register management. Standard optimizations, such as loop interchange, unroll, and unroll-and-jam, can be employed to deal with this problem. In COFFEE, these optimizations are supported either by means of explicit code transformations (interchange, unroll-and-jam) or indirectly by delegation to the compiler through standard pragmas (unroll). Tiling at the level of vector registers is an additional feature of COFFEE. Based on the observation that the evaluation of the element matrix can be reduced to a summation of outer products along the `j` and `k` dimensions, a model-driven vector-register tiling strategy can be implemented. If we consider the codes in the various listings and we focus on the body of the test and trial functions loops (`j` and `k`), the computation of the element matrix is abstractly expressible as

$$A_{jk} = \sum_{\substack{x \in B' \subseteq B \\ y \in B'' \subseteq B}} x_j \cdot y_k \qquad j,k = 0,...,2 \tag{4.8}$$

where B is the set of all basis functions (or temporary variables, e.g., `LI_0`) accessed in the kernel, whereas B' and B'' are generic problem-dependent subsets. Regardless of the specific input problem, by abstracting from the presence of all variables independent of both `j` and `k`, the element matrix computation is always reducible to this form. Figure 4.11 illustrates how we can evaluate 16 entries $(j, k = 0, ..., 3)$ of the element matrix using just 2 vector registers, which represent a 4×4 tile, assuming $|B'| = |B''| = 1$. Values in a register are shuffled each time a product is performed. Standard compiler auto-vectorization for both GNU and Intel compilers, instead, executes 4 broadcast operations (i.e., "splat" of a value over all of the register locations) along the outer dimension to perform the calculation. In addition to incurring a larger number of cache accesses, it needs to keep between $f = 1$ and $f = 3$ extra registers to perform the same 16 evaluations when unroll-and-jam is used, with f being the unroll-and-jam factor.

The storage layout of A, however, is incorrect after the application of

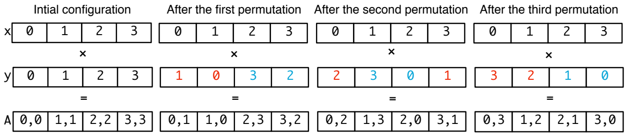
Intial configuration
After the first permutation
After the second permutation
After the third permutation
x
y
A
×
=

Figure 4.11: Outer-product vectorization by permuting values in a vector register.

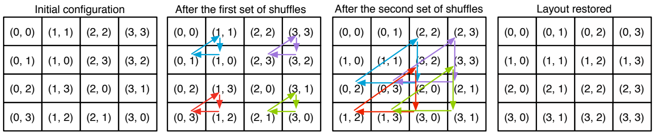
Initial configuration
After the first set of shuffles
After the second set of shuffles
Layout restored

Figure 4.12: Restoring the storage layout after *op-vect*. The figure shows how 4×4 elements in the top-left block of the element matrix A can be moved to their correct positions. Each rotation, represented by a group of three same-colored arrows, is implemented by a single shuffle intrinsic.

this outer-product-based vectorization (*op-vect*, in the following). It can be efficiently restored with a sequence of vector shuffles following the pattern highlighted in Figure 4.12, executed once outside of the `ijk` loop nest. The pseudo-code for the Burgers local assembly kernel when using *op-vect* is shown in Listing 4.5.3.

4.5.4 Exposing Matrix-Matrix Multiplications for BLAS Operations

In this section, a way of systematically transforming the local element matrix computation into a sequence of matrix-matrix multiplication operations is discussed.

If such operations could be exposed, highly-optimized dense linear algebra libraries, for instance MKL or ATLAS BLAS, could be used, which would potentially result in notable performance improvements. It is true that the basis functions' size is usually too small to obtain any gain from BLAS routines, which are tuned for large arrays [Shin et al., 2010]; however, this can significantly increase with the order of the method and due to the presence of coefficient functions in the equation. Since our research is an exploration of optimization techniques for generic equations (i.e. nothing is assumed about the order of the method and the mathematical structure of the form), an algorithm capable of translating assembly expressions into a sequence of BLAS calls has been studied. The main steps of the algorithm are informally provided next.

By fully applying the rewrite rules in Figure 4.7, an assembly expression is reduced to a summation, over each quadrature point, of outer products along the test and trial functions. Each outer product is then isolated, i.e. the assembly expression is split into chunks, each chunk representing an outer product over test and trial functions. Statements in the bodies of the surrounding loops (e.g. coefficients evaluation at a quadrature point, temporaries introduced by expression rewriting) are vector-expanded and hoisted completely outside of the loop nest, similarly to what we have described in Section **??**. This renders the loop nest perfect; that is, there is no intervening code among the various loops. The element matrix evaluation has now become a sequence of dense matrix-matrix multiplies (transposition aside)

$$A_{jk} = \sum_{i} x_{0_{ij}} \cdot y_{0_{ik}} + \sum_{i} x_{1_{ij}} \cdot y_{1_{ik}} + ...$$

where $x0, x1, y0, y1, ...$ are tabulated basis functions or vector-expanded temporaries introduced at expression rewriting time. Eventually, the storage layout of the involved operands is changed so as to be conforming to the BLAS interface (e.g. two dimensional arrays are flatten as one dimensional arrays). The translation into a sequence of DGEMM calls is the last, straightforward step.

4.6 General-purpose Optimizations

4.7 Design and Implementation of COFFEE

4.7.1 Input ad Output: the Integration with Firedrake

4.7.2 Structure

4.7.3 Conveying Domain-Specific Knowledge

4.7.4 Model-driven Dynamic Autotuning

Determining the sequence of transformations that maximizes the performance of a problem requires investigating a broad range of factors, including mathematical structure of the input form, polynomial order of employed function spaces, presence of pre-multiplying functions, and, of course, the characteristics of the underlying architecture. The set of possible optimizations is very large, so the selection problem is challenging. The sole Expression Rewriter, for instance, can generate a wide variety of implementations by just applying rewrite rules up to different extents.

We tackle the optimizations selection problem by compiler autotuning. Not only does it allow to determine the best combination of transformations, also it enables exploring parametric low-level optimizations, such as loop unroll, unroll-and-jam, and interchange, by trying different unroll factors and loop permutations. By leveraging the cost model defined in our previous study, domain knowledge, and a set of heuristics, we manage to keep the autotuner overhead at a minimum, whilst achieving significant speed ups over the purely cost-model-based implementations in all of the forms we have evaluated. In particular, our autotuner usually requires order of seconds to determine the fastest kernel implementation, a negligible overhead when it comes to iterate over real-life unstructured meshes.

COFFEE analyzes the input problem and decides what variants it is worth testing through autotuning, as described later. Each variant is obtained by requesting specific transformations to the Expression Rewriter and the Code Specializer. The possible variants are then provided to the autotuner, in the form of abstract syntax trees. The autotuner is a template-based code generator. By inspecting an abstract syntax tree, it determines how to generate “wrapping” code that 1) initializes kernel’s input variables with fictitious values and 2) calls the kernel. These two points are executed repeatedly in a *while* loop for a pre-established amount of time (order of milliseconds). At the exit of the *while* loop, the times the kernel was invoked is recorded. Eventually, the variant executed the largest number of iterations is designated as the fastest implementation. Suitable compiler directives are used to prevent inlining of all function calls: this avoids the situation in which some variants are inlined and some are not, which would fake the autotuner's output.

The autotuning process is dynamic: depending on the complexity of the input problem, more or less variants are tried. General heuristics, which can be considered a revisited version of those presented in Shin et al. [2010], are applied

- Loop permutations that are likely to worsen the performance are excluded from the search space. According to the cost model, and for the same reasons explained in **?**, we enable only variants in which the loop over quadrature points is either the outermost or the innermost. This is due to the fact that versions of the code in which such loop lies between the test and trial functions loops are typically lower performing.
- The unroll factors must divide the loop bounds evenly to avoid the introduction of reminder (scalar) loops.
- The innermost loop is never explicitly unrolled. This is because we expect auto-vectorization along this loop, so memory accesses should be kept unit-stride.

The autotuner is also domain-aware: the following heuristics, which capture properties of the computational domain, are exploited

- The lengths of test and trial functions loops are identical in some cases, for example when they originate from the same function space. In such cases, since for the employed storage layout the memory access pattern is symmetrical along these two loops, we prune their interchange from the search space.
- The larger is the polynomial order of the method, the larger is the assembly loop nest. In these cases, we impose a bound on the loop nest's

overall unroll factor (which we found empirically) to avoid uselessly testing too many unroll factors.

- On the other hand, if the polynomial order is low, i.e. when the loop nest is small, we prune variants that we know will be low-performing, e.g. those resorting to BLAS.

- We specifically select two levels of expression rewriting. In the "base" level, only generalized loop-invariant code motion, as described in **?**, is applied. This means that only a subset of the rewrite rules exposed in 4.7 will be considered. In the "aggressive" level, all of the rewrite rules are applied. Many other trade-offs, which we do not explore, would be feasibile, however.

- For the expression splitting optimization described in **?** and summarized in Section **??**, we test only three split factors, namely 1, 2, 4. Also, if the input problem uses mixed function spaces, the iteration space is already split at expression rewriting time to avoid computation over zero-valued columns; in these cases, we do not further apply expression splitting.

- Based on the cost model, the padding and data alignment optimization is always applied.

All of the previous points contribute to minimize the overhead of the autotuning process. We will discuss the actual cost of autotuning in Section 4.8.3.

4.8 Performance Analysis

4.8.1 Contribution of Individual Optimizations

4.8.2 Evaluation in Forms of Increasing Complexity

4.8.3 Details on the Autotuning Process

4.8.4 Full Application Study

4.9 Related Word

4.10 Generality of the Approach and Applicability to Other Domains

4.11 Conclusion

Chapter 5

Conclusion

5.1 bho

Bibliography

M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells. Unified Form Language: A domain-specific language for weak formulations of partial differential equations. *ACM Trans Math Software*, 40(2):9:1–9:37, 2014. doi: 10.1145/2566630. URL `http://dx.doi.org/10.1145/2566630`.

AMCG. *Fluidity Manual.* Applied Modelling and Computation Group, Department of Earth Science and Engineering, South Kensington Campus, Imperial College London, London, SW7 2AZ, UK, version 4.0-release edition, November 2010. available at `http://hdl.handle.net/10044/1/7086` .

Krzysztof Bana, Przemyslaw Plaszewski, and Pawel Maciol. Numerical integration on gpus for higher order finite elements. *Comput. Math. Appl.*, 67(6):1319–1344, April 2014. ISSN 0898-1221. doi: 10.1016/j.camwa.2014.01.021. URL `http://dx.doi.org/10.1016/j.camwa.2014.01.021`.

Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375595. URL `http://doi.acm.org/10.1145/1375581.1375595`.

Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. Liszt: A domain specific language for building portable mesh-based pde solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1–9:12, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0771-0. doi: 10.1145/2063384.2063396. URL `http://doi.acm.org/10.1145/2063384.2063396`.

Firedrake contributors. The Firedrake Project. `http://www.firedrakeproject.org`, 2014.

Robert C. Kirby, Matthew Knepley, Anders Logg, and L. Ridgway Scott. Optimizing the evaluation of finite element matrices. *SIAM J. Sci. Comput.*, 27(3):741–758, October 2005. ISSN 1064-8275. doi: 10.1137/040607824. URL `http://dx.doi.org/10.1137/040607824`.

A. Klöckner, T. Warburton, J. Bridge, and J. S. Hesthaven. Nodal discontinuous galerkin methods on graphics processors. *J. Comput. Phys.*, 228(21):7863–7882, November 2009. ISSN 0021-9991. doi: 10.1016/j.jcp.2009.06.041. URL `http://dx.doi.org/10.1016/j.jcp.2009.06.041`.

Matthew G. Knepley and Andy R. Terrel. Finite element integration on gpus. *ACM Trans. Math. Softw.*, 39(2):10:1–10:13, February 2013. ISSN 0098-3500. doi: 10.1145/2427023.2427027. URL `http://doi.acm.org/10.1145/2427023.2427027`.

Anders Logg, Kent-Andre Mardal, Garth N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method.* Springer, 2012. ISBN 978-3-642-23098-1. doi: 10.1007/978-3-642-23099-8.

G. R. Markall, F. Rathgeber, L. Mitchell, N. Loriant, C. Bertolli, D. A. Ham, and P. H. J. Kelly. Performance portable finite element assembly using PyOP2 and FEniCS. In *Proceedings of the International Supercomputing Conference (ISC) '13*, volume 7905 of *Lecture Notes in Computer Science*, June 2013. In press.

Kristian B. Olgaard and Garth N. Wells. Optimizations for quadrature representations of finite element tensors through automated code generation. *ACM Trans. Math. Softw.*, 37(1):8:1–8:23, January 2010. ISSN 0098-3500. doi: 10.1145/1644001.1644009. URL `http://doi.acm.org/10.1145/1644001.1644009`.

Diego Rossinelli, Babak Hejazialhosseini, Panagiotis Hadjidoukas, Costas Bekas, Alessandro Curioni, Adam Bertsch, Scott Futral, Steffen J. Schmidt, Nikolaus A. Adams, and Petros Koumoutsakos. 11 pflop/s simulations of cloud cavitation collapse. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 3:1–3:13, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2378-9. doi: 10.1145/2503210.2504565. URL `http://doi.acm.org/10.1145/2503210.2504565`.

Francis P. Russell and Paul H. J. Kelly. Optimized code generation for finite element local assembly using symbolic manipulation. *ACM Transactions on Mathematical Software*, 39(4), 2013.

Jaewook Shin, Mary W. Hall, Jacqueline Chame, Chun Chen, Paul F. Fischer, and Paul D. Hovland. Speeding up nek5000 with autotuning and specialization. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 253–262, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0018-6. doi: 10.1145/1810085.1810120. URL `http://doi.acm.org/10.1145/1810085.1810120`.

Peter E. J. Vos, Spencer J. Sherwin, and Robert M. Kirby. From h to p efficiently: Implementing finite and spectral/hp element methods to achieve optimal performance for low- and high-order discretisations. *J. Comput. Phys.*, 229(13):5161–5181, July 2010. ISSN 0021-9991. doi: 10.1016/j.jcp.2010.03.031. URL `http://dx.doi.org/10.1016/j.jcp.2010.03.031`.