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

List of Tables

List of Figures

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

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

expected to understand what local assembly represents and how an implementation can be derived starting from a mathematical specification of the finite element problem.

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 Optimizing Finite Element Local Assembly

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.

4.4.1 Generalized Loop-invariant Code Motion

4.4.2 Terms Factorization

4.4.3 Precomputation of Invariant Terms

4.4.4 Expanding Sub-expressions

4.4.5 Rewrite Rules of Assembly Expressions

4.4.6 Avoiding Iteration over Zero-valued Blocks by Symbolic Execution

4.5 Code Specialization

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

4.8 Performance Analysis

4.8.1 Contribution of Individual Optimizations

4.8.2 Evaluation in Forms of Increasing Complexity

4.8.3 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.

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.